Библиотека профессионала

Java™ 2

Том 2. Тонкости программирования

ВОСЬМОЕ ИЗДАНИЕ

Java_2_title.indd 1

Core Java

Volume II – Advanced Features

Eighth Edition

Cay S. Horstmann
Gary Cornell



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

Java_2_title.indd 2

Библиотека профессионала

Java™ 2

Том 2. Тонкости программирования

Восьмое издание

Кей Хорстманн
Гари Корнелл



Москва • Санкт-Петербург • Киев
2012

Java_2_title.indd 3 03.10.2008 12:46:38

ББК 32.973.26-018.2.75
Х82
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С. Н. Тригуб*
Перевод с английского *Я. П. Волковой, Д. Я. Иваненко*
Под редакцией *Ю. Н. Артеменко*
По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Хорстманн, Кей С., **Корнелл**, Гари.

Х82 Java 2. Библиотека профессионала, том 2. Тонкости программирования, 8-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2012. — 992 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1482-8 (рус.)

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, Ptr., Copyright © 2008 Sun Microsystems, Inc.

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.

Sun Microsystems, Inc., has intellectual property rights relating to implementations of the technology described in this publication. In particular, and without limitation, these intellectual property rights may include one or more U.S. patents, foreign patents, or pending applications. Sun, Sun Microsystems, the Sun logo, J2ME, Solaris, Java, Javadoc, NetBeans, and all Sun and Java based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

Figures 7–45, 7–46: "Foxkeh" © 2006 Mozilla Japan.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2012

Научно-популярное издание
Кей С. Хорстманн, Гари Корнелл

Java 2. Библиотека профессионала, том 2 Тонкости программирования, 8-е издание

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

Подписано в печать 03.05.2012. Формат 70×100/16
Гарнитура Times. Печать офсетная
Усл. печ. л. 79,98. Уч.-изд. л. 55,95
Доп. тираж 1000 экз. Заказ № 0000

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 978-5-8459-1482-8 (рус.)
ISBN 978-01-3235479-0 (англ.)

© Издательский дом "Вильямс", 2012
© Sun Microsystems, Inc., 2008

Оглавление

Предисловие 15

Глава 1. Потоки и файлы 19

Глава 2. Язык XML 103

Глава 3. Работа в сети 181

Глава 4. Работа с базами данных 227

Глава 5. Интернационализация 305

Глава 6. Расширенные средства Swing 355

Глава 7. Расширенные средства AWT 515

Глава 8. Компоненты JavaBeans 669

Глава 9. Средства защиты 737

Глава 10. Работа с удаленными объектами 823

Глава 11. Написание сценариев, компиляция кода и обработка аннотаций 865

Глава 12. Платформенно-ориентированные методы 915

Предметный указатель 971

Содержание

Об авторах 13
Благодарности 13

Предисловие 15

К читателю 15
Краткий тур по книге 15
Соглашения 17
Коды примеров 18
От издательства 18

Глава 1. Потоки и файлы 19

Потоки 20
Чтение и запись байтов 20
Коллекция потоков 22
Комбинирование потоковых фильтров 26
Текстовые входные и выходные данные 29
Запись текстовых выходных данных 30
Считывание текстовых входных данных 32
Сохранение объектов в текстовом формате 32
Наборы символов 37
Чтение и запись двоичных данных 41
Файлы с произвольным доступом 43
ZIP-архивы 49
Потоки объектов и сериализация 55
Понимание формата файлов, создаваемых при сериализации объектов 61
Изменение механизма сериализации, используемого по умолчанию 66
Сериализация одноэлементных множеств и перечислений, обеспечивающих безопасность типов 68
Управление версиями 69
Использование сериализации для клонирования 71
Управление файлами 73
Новые средства для обработки ввода-вывода 80
Отображаемые в памяти файлы 81
Структура буферных данных 87
Блокирование файлов 89
Регулярные выражения 91

Глава 2. Язык XML 103

Введение в XML 104
Структура XML-документа 106
Разбор XML-документа 109
Проверка XML-документов 120
Определения типов документов 121
XML-схема 128
Практический пример XML-документа 130
Поиск информации с помощью XPath 144

Использование пространств имен 150
Потоковые анализаторы 152
Использование SAX-анализатора 152
Использование StAX-анализатора 157
Генерирование XML-документов 160
Создание XML-документа посредством StAX 164
XSLT-преобразования 171

Глава 3. Работа в сети 181

Подключение к серверу 182
Тайм-ауты сокетов 186
Internet-адреса 187
Реализация серверов 189
Обслуживание множества серверов 191
Одностороннее закрытие 195
Прерываемые сокеты 196
Отправка электронной почты 202
Создание соединений с URL 207
`URL` и `URI` 207
Извлечение информации с помощью `URLConnection` 208
Отправка данных формы 218

Глава 4. Работа с базами данных 227

Структура JDBC 228
Типы JDBC-драйверов 229
Типичные примеры использования JDBC 230
Язык SQL 231
Конфигурирование JDBC 236
URL базы данных 237
JAR-файлы драйверов 237
Запуск базы данных 237
Регистрация класса драйвера 238
Соединение с базой данных 239
Выполнение операторов SQL 242
Управление соединениями, командами и результирующими наборами 244
Анализ SQL-исключений 245
Заполнение базы данных 247
Выполнение запроса 251
Предварительно подготовленные команды 252
Чтение и запись больших объектов 259
Синтаксис перехода в SQL 261
Множественные результаты 262
Получение автоматически генерируемых ключей 263
Прокручиваемые и обновляемые результирующие наборы 264
Прокрутка результирующего набора 264
Обновляемые результирующие наборы 266
Наборы строк 270
Кэшируемые наборы строк 270
Метаданные 273

Транзакции 282
Точки сохранения 283
Пакетные обновления 283
Расширенные типы данных в языке SQL 285
Управление соединениями в Web- и производственных приложениях 287
Введение в LDAP 288
Настройка LDAP-сервера 289
Доступ к каталогу LDAP 292

Глава 5. Интернационализация 305

Региональные стандарты 306
Форматы чисел 311
Валюты 316
Дата и время 318
Сортировка 325
Избирательность сортировки 326
Декомпозиция 327
Форматирование сообщений 332
Форматы выбора 334
Текстовые файлы и наборы символов 336
Кодирование символов в файлах исходного кода 336
Пакеты ресурсов 337
Местонахождение пакетов ресурсов 337
Файлы свойств 338
Классы пакетов 339
Пример приложения 341

Глава 6. Расширенные средства Swing 355

Списки 356
Компонент `JList` 356
Модели списков 362
Вставка и удаление значений 367
Отображение значений 368
Таблицы 373
Простая таблица 374
Модели таблиц 377
Работа со строками и столбцами 381
Деревья 407
Простые деревья 407
Перечисление узлов 423
Визуализация узлов 425
Обработка событий деревьев 427
Специальные модели деревьев 434
Текстовые компоненты 442
Изменение отслеживания в текстовых компонентах 442
Поля форматированного ввода 446
Фильтры 448
Отображение HTML-данных посредством панели `JEditorPane` 469

Получение информации о выполнении задачи 476
Индикатор хода процесса 476
Мониторы задач 479
Отслеживание чтения данных из потока 483
Организаторы компонентов 488
Панели с границей раздела 488
Панели с вкладками 492
Панель рабочего стола и внутренние фреймы 498
Каскадное и мозаичное расположение фреймов 501
Запрет на изменение свойства 504

Глава 7. Расширенные средства AWT 515

Алгоритм рисования 516
Фигуры 518
Классы фигур 520
Области 532
Штрихи 533
Расцветки 541
Преобразование координат 543
Отсечение 547
Прозрачность и композиция 549
Параметры рисования 558
Чтение и запись изображений 564
Получение объектов для чтения и записи графических файлов 564
Чтение и запись файлов, содержащих несколько изображений 566
Управление изображениями 574
Создание растровых изображений 574
Фильтрация изображений 581
Печать 590
Вывод на печать графических изображений 590
Многостраничная печать 599
Предварительный просмотр 601
Службы печати 610
Потоковые службы печати 614
Атрибуты печати 614
Буфер обмена 621
Классы и интерфейсы для передачи данных 622
Передача текста 623
Интерфейс Transferable и типы документов 627
Передача изображений через буфер обмена 629
Передача объектов Java через системный буфер обмена 634
Использование локального буфера обмена для передачи ссылок на объекты 638
Копирование путем перетаскивания 638
Поддержка передачи данных в компонентах Swing 640
Источники перетаскивания 644
Приемники перетаскивания 646
Средства интеграции в платформу 654
Начальные экраны-заставки 654
Запуск настольных приложений 659
Системный лоток 664

Глава 8. Компоненты JavaBeans 669

Зачем нужны компоненты JavaBeans 670
Создание компонентов JavaBeans 672
Применение компонентов JavaBeans для создания приложений 674
Упаковка компонентов JavaBeans в JAR-файлах 675
Создание компонентов JavaBeans в интегрированной среде разработки 676
Соглашения об именовании свойств и событий компонента JavaBeans 682
Типы свойств компонентов JavaBeans 684
Простые свойства 684
Индексированные свойства 685
Связанные свойства 685
Свойства с ограничениями 687
Классы `BeanInfo` 693
Редакторы свойств 697
Создание редакторов свойств 700
Строковые редакторы свойств 700
Визуальные редакторы свойств 703
Настройщики 707
Создание класса настройщика 709
Постоянные компоненты JavaBeans 715
Использование средств обеспечения постоянства JavaBeans для работы с произвольными данными 719
Использование делегатов при создании объектов 719
Целый пример постоянного компонента JavaBeans 724

Глава 9. Средства защиты 737

Загрузчики классов 738
Иерархия загрузчиков классов 740
Использование загрузчиков классов в качестве пространств имен 742
Создание собственного загрузчика классов 742
Верификация байтового кода 748
Диспетчеры защиты и полномочия 753
Модель защиты Java 754
Файлы политики безопасности 758
Пользовательские права доступа 765
Реализация класса полномочий 766
Аутентификация пользователей 771
Регистрационные модули JAAS 777
Цифровые подписи 786
Дайджест сообщения 787
Подписание сообщений 793
Формат сертификата X.509 795
Проверка подписи 797
Проблема аутентификации 799
Подписание сертификатов 801
Запросы сертификатов 802
Подписание кода 803
Подписание JAR-файлов 804
Сертификаты разработчиков программ 808

Шифрование 810
Симметричные шифры 810
Генерация ключей 812
Потоки шифрования 816
Шифрование с открытым ключом 818

Глава 10. Работа с удаленными объектами 823

Клиент и сервер 824
Удаленные вызовы методов 827
Заглушки и передача параметров 827
Модель программирования RMI 829
Интерфейсы и реализации 829
Служба регистрации RMI 830
Развертывание программы 834
Регистрация активности RMI 837
Параметры и возвращаемые значения в удаленных методах 839
Передача удаленных объектов 839
Передача не удаленных объектов 839
Динамическая загрузка классов 841
Удаленные ссылки с множеством интерфейсов 846
Удаленные объекты и методы `equals()`, `hashCode()` и `clone()` 847
Активизация удаленных объектов 847
Web-службы и JAX-WS 853
Использование JAX-WS 853
Клиент Web-службы 856
Служба электронного магазина Amazon 859

Глава 11. Написание сценариев, компиляция кода и обработка аннотаций 865

Написание сценариев для платформы Java 866
Получение механизма сценариев 866
Оценка сценариев и привязки 867
Перенаправление входных и выходных данных 869
Вызов сценарных функций и методов 870
Компиляция сценария 872
Пример: создание сценария обработки событий GUI-интерфейса 873
API-интерфейс компилятора 877
Компилирование простым путем 877
Использование заданий на компиляцию 877
Пример: Динамическая генерация кода Java 882
Использование аннотаций 887
Пример: аннотирование обработчиков событий 888
Синтаксис аннотаций 893
Стандартные аннотации 897
Аннотации для компиляции 897
Аннотации для управления ресурсами 898
Мета-аннотации 898
Обработка аннотаций на уровне исходного кода 900
Разработка байт-кода 907
Изменение байт-кодов во время загрузки 912

Глава 12. Платформенно-ориентированные методы 915
Вызов функции на языке C из Java-программы 916
Числовые параметры и возвращаемые значения 922
Использование функции `printf()` для форматирования чисел 922
Строковые параметры 924
Получение доступа к полям объектов 929
Получение доступа к полям экземпляра 929
Получение доступа к статическим полям 933
Кодирование сигнатуры 934
Вызов Java-методов 935
Методы экземпляра 936
Статические методы 937
Конструкторы 938
Альтернативные вызовы методов 938
Доступ к элементам массивов 942
Обработка ошибок 945
API вызова 950
Пример обращения к реестру Windows 955
Общие сведения о реестре Windows 955
Java-интерфейс для доступа к реестру 956
Организация доступа к реестру посредством платформенно-ориентированных методов 957

Предметный указатель 971

Об авторах

Кей Хорстманн (Cay Horstmann) — профессор факультета вычислительной техники в Университете Сан-Хосе, обладатель звания "Чемпион по Java" и частый докладчик на многих отраслевых конференциях. Является соавтором книги *JavaServer Faces. Библиотека профессионала* (ИД "Вильямс", 2008 г.).

Гари Корнелл (Gary Cornell) имеет более чем 20-летний опыт обучения и написания книг по программированию. Является одним из основателей издательства Apress. Он написал множество книг-бестселлеров для профессионалов, попал в финал конкурса Jolt Award, а также стал обладателем звания "Выбор читателей" в журнале *Visual Basic Magazine*.

Благодарности

Написание книги всегда требует приложения огромных усилий, да и переделка тоже, похоже, не дается намного легче, особенно при такой высокой скорости развития технологии Java. Выход книги в свет подразумевает участие массы людей, и я с большим удовольствием хотел бы поблагодарить всех членов команды, работавших над этой книгой.

Многие сотрудники из издательства Prentice Hall и Sun Microsystems Press оказали ценную помощь и при этом остались незамеченными. Я хочу, чтобы они знали, что я очень благодарен им за их усилия. Как обычно, искренние слова благодарности моему издателю, Грегу Доэнчу (Greg Doench), из Prentice Hall за управлением процессом написания и изготовления книги и позволение мне пребывать в благодатном неведении обо всех тех, кто за этим стоит. Благодарю Ванессу Мур (Vanessa Moore) за замечательную поддержку в вопросах производства. Отдельное спасибо моему соавтору по предыдущим изданиям Гари Корнеллу (Gary Cornell), который после этого погрузился уже в другие приключения.

Спасибо многочисленным читателям предыдущих выпусков за предоставление информации о допущенных ошибках и привнесение массы стоящих предложений по улучшению. Особую благодарность хотелось бы выразить команде рецензентов, которые тщательнейшим образом изучили все детали данного манускрипта и избавили меня от массы других глупых ошибок.

В рецензировании этого и предыдущих изданий участие принимали: Чак Эллисон (Chuck Allison, ассистент редактора в журнале *C/C++ Users Journal*), Лэнс Андерсон (Lance Anderson, из Sun Microsystems), Алек Битон (Alec Beaton, из PointBase, Inc.), Клиф Берг (Cliff Berg, из iSavvix Corporation), Джошуа Блок (Joshua Bloch, из Sun Microsystems), Дэвид Браун (David Brown), Корки Картрайт (Corky Cartwright), Фрэнк Коэн (Frank Cohen, из PushToTest), Крис Крэйн (Chris Crane, из devXsolution), доктор Николас де Лилло (Dr. Nicholas J. De Lillo, из колледжа Манхеттена), Ракеш Дхупар (Rakesh Dhoopar, из Oracle), Роберт Эванс (Robert Evans, ведущий специалист из лаборатории прикладной физики при университете Джона Хопкинса), Дэвид Гери (David Geary, из Sabreware), Брайан Гоэтс (Brian Goetz, главный консультант из Quiotix Corp.), Анжела Гордон (Angela Gordon, из Sun Microsystems), Дэн Гордон (Dan Gordon, из Sun Microsystems), Роб Гордон (Rob Gordon), Джон Грэй (John Gray, из университета Хартфорда), Камерон Грегори (Cameron Gregory, из olabs.com), Марти Холл (Marty Hall, из лаборатории прикладной физики при университете Джона Хопкинса), Винсент Харди (Vincent Hardy, из Sun Microsystems), Дэн Харки (Dan Harkey, из Государственного университета в Сан-Хосе), Уильям Хиггинс (William Higgins, из IBM),

Владимир Иванович (Vladimir Ivanovic, из PointBase), Джерри Джексон (Jerry Jackson, из ChannelPoint Software), Тип Киммет (Tim Kimmet, из Preview Systems), Крис Лаффра (Chris Laffra), Чарли Лаи (Charlie Lai, из Sun Microsystems), Анжелика Лэнгер (Angelika Langer), Дуг Лэнгстон (Doug Langston), Хэнг Лау (Hang Lau, из университета имени Мак-Гилла), Марк Лоуренс (Mark Lawrence), Дуг Ли (Doug Lea, из SUNY Oswego), Грегори Лонгшор (Gregory Longshore), Боб Линч (Bob Lynch, из Lynch Associates), Филип Милн (Philip Milne, консультант), Марк Морриссей (Mark Morrissey, из Выпускного института штата Орегон), Махеш Нилаканта (Mahesh Neelakanta, из Атлантического университета, штат Флорида), Хао Фам (Hao Pham), Пол Филион (Paul Philion), Блейк Регсделл (Blake Ragsdell), Илбер Рамадани (Ylber Ramadani, из университета Райерсона), Стюарт Регес (Stuart Reges из Университета Аризоны), Рич Розен (Rich Rosen, из Interactive Data Corporation), Питер Сандерс (Peter Sanders, из университета ESSI, Ницца, Франция), доктор Пол Сангера (Dr. Paul Sanghera, из Государственного университета в Сан-Хосе и колледжа в Бруксе), Пол Севинк (Paul Sevinc, из Teamup AG), Деванг Ша (Devang Shah, из Sun Microsystems), Ричард Сливчак (Richard Slywczak, из исследовательского центра NASA/Glenn), Брэдли Смит (Bradley A. Smith), Стивен Стелтинг (Steven Stelting, из Sun Microsystems), Кристофер Тэйлор (Christopher Taylor), Люк Тэйлор (Luke Taylor, из Valtech), Джордж Тируватукал (George Thiruvathukal), Ким Топли (Kim Topley, автор книги *Core JFC*), Джанет Трауб (Janet Traub), Пол Тима (Paul Tyma, консультант), Питер Ван дер Линден (Peter van der Linden, из Sun Microsystems) и Берт Уолш (Burt Walsh).

Кей Хорстманн
Сан-Франциско, 2008 г.

ПРЕДИСЛОВИЕ

К читателю

Книгу, которую вы сейчас держите в своих руках, является вторым томом восьмого издания книги, которое полностью адаптировано под Java SE 6. В первом томе рассказывалось о базовых функциональных возможностях языка Java; в этом же томе рассматриваются более сложные темы, которые могут понадобиться программисту для профессиональной разработки программного обеспечения. Поэтому, как и в первом томе и предыдущих изданиях настоящей книги, здесь по-прежнему расчет делается на тех программистов, которые планируют применять технологию Java для работы над реальными проектами.

Просим обратить внимание, что опытным разработчикам, умеющим работать с такими сложными средствами языка, как внутренние и обобщенные классы, читать первый том для понимания содержимого второго тома, вовсе не обязательно. Несмотря на то что в этом томе, конечно же, где необходимо, встречаются ссылки на разделы предыдущего тома (который, мы надеемся, вы все-таки приобретете или уже приобрели), весь необходимый базовый материал можно легко найти в любой более или менее полной вводной книге по платформе Java.

Краткий тур по книге

Главы в этой книге по большей части не зависят друг от друга, что дает читателю возможность начинать изучение материала с той темы, которая интересует его больше всего, и вообще читать все главы в любом удобном ему порядке.

- **Глава 1** посвящена обработке входных и выходных данных. В Java весь ввод и вывод обрабатывается посредством так называемых *потоков* (streams). Потоки позволяют однотипным образом иметь дело с передачей данных между различными источниками данных, такими как файлы, сетевые соединения и блоки памяти. Эта глава сначала включает детальное описание классов чтения (`Reader`) и записи (`Writer`), которые облегчают работу с Unicode. Далее в ней демонстрируются процессы, происходящие внутри в случае применения механизма сериализации объектов, который делает сохранение и загрузку объектов простой и удобной. И, наконец, в заключительной части этой главы рассказывается о классах "новых средств для обработки ввода-вывода" (являвшихся новыми на тот момент, когда они добавлялись в Java SE 1.4), которые поддерживают эффективные файловые операции, а также о библиотеке регулярных выражений.
- **Глава 2** посвящена XML. Здесь показывается, как выполнять синтаксический анализ файлов XML, как генерировать XML и как использовать трансформации XSL. В качестве примера описывается процесс определения компоновки формы Swing из XML. Новым дополнением в этой главе является рассмотрение API-интерфейса XPath, который значительно упрощает поиск мелких деталей в больших объемах XML.
- **Глава 3** посвящена сетевому API-интерфейсу. Язык Java позволяет феноменально легко выполнять сложные операции по программированию сетевых задач. В этой главе демонстрируется, как создавать сетевые соединения с серверами, как реализовывать свои собственные серверы и как открывать HTTP-подключения.

- **Глава 4** посвящена программированию баз данных. Основное внимание здесь уделяется интерфейсу JDBC (Java Database Connectivity API — API-интерфейс для организации доступа к базам данных из Java-приложений), который позволяет Java-программам подключаться к реляционным базам данных. Помимо этого здесь также показывается, как писать полезные программы для обработки реальных рутинных операций баз данных с применением только базового поднабора средств API-интерфейса JDBC. (Для рассмотрения всех средств API-интерфейса JDBC потребовалась бы книга почти такого же размера, как эта.) В завершение главы приводится краткая информация об иерархических базах данных, а также об интерфейсе JNDI (Java Naming and Directory Interface — интерфейс именования и каталогов Java) и протоколе LDAP (Lightweight Directory Access Protocol — облегченный протокол доступа к каталогам).
- **Глава 5** посвящена такой технологии, как интернационализация, важность которой, по нашему мнению, будет только возрастать. Язык программирования Java является одним из нескольких языков, которые с самого начала предусматривали возможность обработки Unicode, но поддержка интернационализации в Java позволяет идти гораздо дальше. В частности, путем интернационализации приложений Java разработчик может делать так, чтобы они пересекали границы не только платформы, но и страны. Например, в этой главе демонстрируется, как можно написать аплет для вычисления времени выхода на пенсию, применяющий английский, немецкий или китайский языки, в зависимости от выбранных в браузере региональных и языковых настроек.
- **В главе 6** содержится весь материал по Swing, который не вошел в первый том, в частности, описание важных и сложных компонентов деревьев и таблиц. Здесь демонстрируются основные способы применения панелей редактора, Java-реализация многодокументного интерфейса, индикаторы хода выполнения, которые используются в многопоточных программах, и "средства интеграции для рабочего стола" вроде заставок и поддержки для области пиктограмм панели задач. Опять-таки, основное внимание уделяется только наиболее полезным конструкциям, с которыми разработчикам наверняка доведется встречаться на практике, поскольку даже краткое рассмотрение всей библиотеки Swing заняло бы целых несколько томов и представляло бы интерес только для очень узких специалистов.
- **Глава 7** посвящена API-интерфейсу Java 2D, которым можно пользоваться для создания реалистичных изображений и специальных эффектов. В ней также рассказывается и о некоторых более совершенных средствах AWT (Abstract Windowing Toolkit — набор инструментальных средств для абстрактных окон), которые оказались слишком специфическими для рассмотрения в первом томе, но которые, тем не менее, должны присутствовать в наборе инструментов каждого программиста. К их числу относятся средства печати и API-интерфейсы для обеспечения возможности выполнения операций вырезания, вставки и перетаскивания.
- В **главе 8** рассказывается обо всем, что разработчикам необходимо знать о доступном для платформы Java API-интерфейсе программирования компонентов под названием JavaBeans. Здесь показано, как создавать свои собственные bean-компоненты так, чтобы другие программисты могли манипулировать ими в интегрированных средах разработки. Завершается глава описанием способов применения предлагаемого в Java механизма постоянства для сохранения данных в таком формате, который, в отличие от получаемого при сериализации объектов, подходит для длительного хранения.

- **Глава 9** посвящена модели безопасности Java. Платформа Java с самого начала разрабатывалась с учетом безопасности, и в этой главе объясняется, что именно позволяет ей эту безопасность обеспечивать. Первым делом здесь демонстрируется, как создавать свои собственные загрузчики классов и диспетчеры защиты для специальных приложений. Далее рассказывается об API-интерфейсе безопасности, который позволяет оснащать приложения важными средствами, вроде механизма создания цифровых подписей для сообщений и кода, механизма авторизации и аутентификации и механизма шифрования. Завершается глава демонстрацией примеров с применением таких алгоритмов шифрования, как AES и RSA.
- **Глава 10** посвящена распределенным объектам. В частности, здесь подробно рассматривается протокол RMI (Remote Method Invocation — удаленный вызов методов). Этот протокол, по сути, представляет собой API-интерфейс и позволяет работать с Java-объектами, которые разбросаны по множеству компьютеров. Кроме того, здесь еще также вкратце рассказывается о Web-службах и приводится пример, в котором программа Java взаимодействует с Web-службой Amazon.
- В **главе 11** описываются три различных технологии для обработки кода. API-интерфейсы сценариев и компилятора, представленные в версии Java SE 6, позволяют программе вызывать код, написанный на каком-то языке сценариев, например, JavaScript или Groovy, и компилировать код на языке Java. Аннотации позволяют добавлять в Java-программу произвольно выбираемую информацию (еще иногда также называемую метаданными). В этой главе показывается, как обработчики аннотаций могут собирать аннотации на уровне источника и на уровне файлов классов, и как аннотации могут применяться для оказания влияния на поведение классов во время выполнения. Аннотации выгодно использовать только вместе с соответствующими инструментальными средствами, и мы надеемся, что наш материал поможет читателю научиться выбирать именно те средства обработки аннотаций, которые наиболее отвечают его потребностям.
- **Глава 12** посвящена платформенно-ориентированным методам, которые позволяют вызывать методы, предназначенные для конкретной машины, например, для Microsoft Windows API. Очевидно, что данное средство является спорным, ведь использование платформенно-ориентированных методов ведет к исчезновению межплатформенных преимуществ платформы Java. Тем не менее, каждый серьезный программист, занимающийся написанием Java-приложений для конкретных платформ, должен обязательно владеть им. Время от времени все-таки встречаются ситуации, когда требуется переключаться на API-интерфейс операционной системы целевой платформы при взаимодействии с устройством или службой, которая не поддерживается платформой Java. То, как подобное делается, как раз и иллюстрируется в данной главе на примере осуществления из Java-программы доступа к API-интерфейсу системного реестра Windows.

Как обычно, все главы были полностью переделаны под последнюю версию Java. Весь устаревший материал был изъят, а новые API-интерфейсы, появившиеся только в версии Java SE 6, рассматриваются в мельчайших деталях.

Соглашения

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



Такой пиктограммой выделяются замечания.



Такой пиктограммой выделяются советы.



Такой пиктограммой выделяются предупреждения об опасности.



В нашей книге есть множество замечаний, касающихся C++, в которых объясняются отличия между языками Java и C++. Вы можете пропустить их, если не имеете опыта работы с C++ или если склонны воспринимать этот опыт как страшный сон, который лучше забыть.



Интерфейс прикладного программирования

Язык Java сопровождается огромной библиотекой интерфейса прикладного программирования (API). Используя вызов API впервые, в конце соответствующего раздела мы добавляем краткое описание, сопровождаемое такой пиктограммой. Эти описания не слишком информативны, но, как мы надеемся, более информативны, чем те, что представлены в официальной онлайновой документации по API. Мы снабжаем каждую врезку по API номером версии, в которой средство впервые было представлено, чтобы помочь тем читателям, которые не используют самую последнюю версию Java.

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

Листинг 11.1. Содержимое файла `ScriptTest.java`

Коды примеров

Коды примеров, рассмотренных в книге, доступны как на Web-сайте, посвященном книге (`http://horstmann.com/corejava`), так и на Web-сайте издательства.

От издательства

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

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

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

Наши координаты:

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

Информация для писем из:

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

Глава 1

ПОТОКИ И ФАЙЛЫ

- Потоки
- Текстовые входные и выходные данные
- Чтение и запись двоичных данных
- ZIP-архивы
- Потоки объектов и сериализация
- Управление файлами
- Новые средства для обработки ввода-вывода
- Регулярные выражения

В этой главе речь пойдет об интерфейсах прикладного программирования (Application Programming Interface – API), которые доступны в Java для организации ввода и вывода данных. В частности, здесь читатель сможет узнать о том, как получать доступ к файлам и каталогам, а также как считывать и записывать данные в двоичном и текстовом формате. Кроме того, в этой главе еще также будет рассмотрен механизм сериализации, позволяющий сохранять объекты так же легко, как и текстовые или числовые данные. Далее будут перечислены несколько улучшений, которые были добавлены в пакет "новых средств для обработки ввода-вывода" (new I/O – NIO) с соответствующим названием `java.nio`, представленный впервые еще в версии Java SE 1.4. В завершение этой главы мы поговорим о регулярных выражениях, хотя они и не имеют непосредственного отношения к потокам и файлам. Нам не удалось найти лучшего места для обсуждения этой темы, как его, очевидно, не удалось найти и команде разработчиков Java, присоединивших спецификацию API регулярных выражений к запросу спецификации по новым средствам для обработки ввода-вывода в Java SE 1.4.

Потоки

В API-интерфейсе Java объект, из которого можно считывать последовательность байтов, называется *потоком входных данных* (input stream), а объект, в который можно записывать последовательность байтов – *потоком выходных данных* (output stream). В роли таких источников и пунктов назначения для последовательностей байтов чаще всего выступают файлы, но также могут выступать и сетевые подключения или даже блоки памяти. Абстрактные классы `InputStream` и `OutputStream` образуют основу иерархии классов ввода-вывода (input/output – I/O).

Из-за того, что байтовые (byte-oriented) потоки являются неудобными для обработки информации, хранящейся в кодировке Unicode (напоминаем, что в кодировке Unicode на каждый символ приходится несколько байтов), для обработки Unicode-символов предусмотрена отдельная иерархия классов, все из которых наследуются от абстрактных классов `Reader` и `Writer`. Эти классы позволяют выполнять операции чтения и записи на базе не однобайтовых символов, а двухбайтовых единиц кода Unicode.

Чтение и запись байтов

Класс `InputStream` имеет следующий абстрактный метод:

```
abstract int read()
```

Этот метод считывает один байт и возвращает либо считанный байт, либо `-1`, если наталкивается на конец потока входных данных. Разработчик конкретного класса потока входных данных может переопределять этот метод так, чтобы он предоставлял какую-то полезную функциональную возможность. Например, в классе `FileInputStream` этот метод выполняет считывание одного байта из файла. `System.in` представляет собой предопределенный объект подкласса `InputStream`, который позволяет считывать информацию с клавиатуры.

У класса `InputStream` еще также имеются и неабстрактные методы для считывания массива байтов или для пропускания ряда байтов. Эти методы вызывают абстрактный метод `read`, благодаря чему подклассам нужно переопределять только один метод.

Аналогичным образом, класс `OutputStream` определяет следующий абстрактный метод, который записывает один байт в место выходных данных:

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

Как методы `read`, так и методы `write` *блокируют доступ* до тех пор, пока байты действительно не будут считаны или записаны. Это означает, что если к потоку данных не удается получить доступ мгновенно (что обычно происходит из-за занятости сетевого подключения), происходит блокирование текущего потока задач (thread). А это дает другим потокам задач возможность выполнять какую-то полезную работу, пока метод ожидает, когда поток данных снова станет доступным.

Метод `available` позволяет проверять, какое количество байтов доступно для считывания в текущий момент. Это означает, что фрагмент кода, вроде того, что показан ниже, вряд ли будет приводить к блокировке:

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

По завершении процесса считывания или записи данных в поток нужно обязательно закрывать его путем вызова метода `close`. Такой вызов приводит к освобождению системных ресурсов, доступных в ограниченном количестве. В случае открытия приложением множества потоков без последующего их закрытия, ресурсы системы могут истощаться. Помимо этого, закрытие потока выходных данных еще также приводит и к *очистке* (flush) использовавшегося для него буфера: все символы, которые временно помещались в этот буфер с целью их последующей доставки уже сразу в виде бо́льшего пакета, рассылаются по пунктам назначения. В частности, если не закрыть файл, последний пакет байтов может быть так никогда и не доставлен. Выполнять сбрасывание выходных данных также можно и вручную с помощью метода `flush`.

Даже если класс потока предоставляет конкретные методы для работы с исходными функциями `read` и `write`, программисты приложений редко их используют. Для них интерес чаще представляют все-таки данные, содержащие числа, строки и объекты, а не исходные байты.

Поэтому в Java предлагается много классов потоков, которые наследуются от базовых классов `InputStream` и `OutputStream` и позволяют работать с данными именно в том виде, в котором обычно требуется программистам, а не просто на уровне байтов.

API java.io.InputStream 1.0

- `abstract int read()`

  Считывает байт данных и возвращает его. В конце потока этот метод `read` возвращает значение `-1`.

- `int read(byte[] b)`

  Выполняет считывание данных в байтовый массив и возвращает фактическое количество считанных байтов или значение `-1`, если он дошел уже до конца потока. Этот метод `read` может считывать максимум `b.length` байтов.

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

  Выполняет считывание данных в байтовый массив. Этот метод `read` возвращает фактическое количество считанных байтов или значение `-1`, если он уже дошел до конца потока.

  Параметры:
  - `b` Массив, в который должны считываться данные.
  - `off` Позиция (смещение) в `b`, с которой должно начинаться размещение байтов.
  - `len` Максимальное количество подлежащих считыванию байтов.

- `long skip(long n)`

  Пропускает `n` байтов в потоке входных данных. Возвращает фактическое количество пропущенных байтов (которое в случае достижения конца потока может быть и меньше `n`).

- `int available()`

  Возвращает количество байтов, доступных без блокирования. (Напоминаем, что блокирование означает потерю текущим потоком (thread) своей очереди.)

- `void close()`

  Закрывает поток входных данных.

- `void mark(int readlimit)`

  Устанавливает на текущей позиции в потоке входных данных маркер. (Не все потоки поддерживают такую функциональность.) В случае считывания из потока входных данных большего, чем указано в `readlimit`, количества байтов, потоку разрешено забыть о маркере.

- `void reset()`

  Возвращается к последнему маркеру. Последующие вызовы метода `read` приводят к повторному считыванию байтов. В случае отсутствия маркера "сброс" (reset) потока не происходит.

- `boolean markSupported()`

  Возвращает `true`, если поток поддерживает возможность установки маркеров.

API `java.io.OutputStream 1.0`

- `abstract void write(int n)`

  Записывает байт данных.

- `void write(byte[] b)`

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

  Записывает все байты или диапазон байтов из массива `b`.

  Параметры:

`b`	Массив, из которого должны браться данные для записи.
`off`	Позиция (смещение) в массиве `b`, с которой должна начинаться запись первого байта.
`len`	Общее количество байтов, подлежащих записи.

- `void close()`

  Выполняет очистку (flush) и закрывает поток выходных данных.

- `void flush()`

  Выполняет очистку потока выходных данных, т.е. отправляет любые находящиеся в буфере данные в их место назначения.

Коллекция потоков

В отличие от языка C, которому вполне хватает одного единственного типа `FILE*`, в Java имеется целая коллекция, состоящая из более чем 60 (!) различных типов потоков (рис. 1.1 и рис. 1.2).

Давайте разобьем эти типы по сферам их применения. Для классов, обрабатывающих байты и символы, существуют отдельные иерархии.

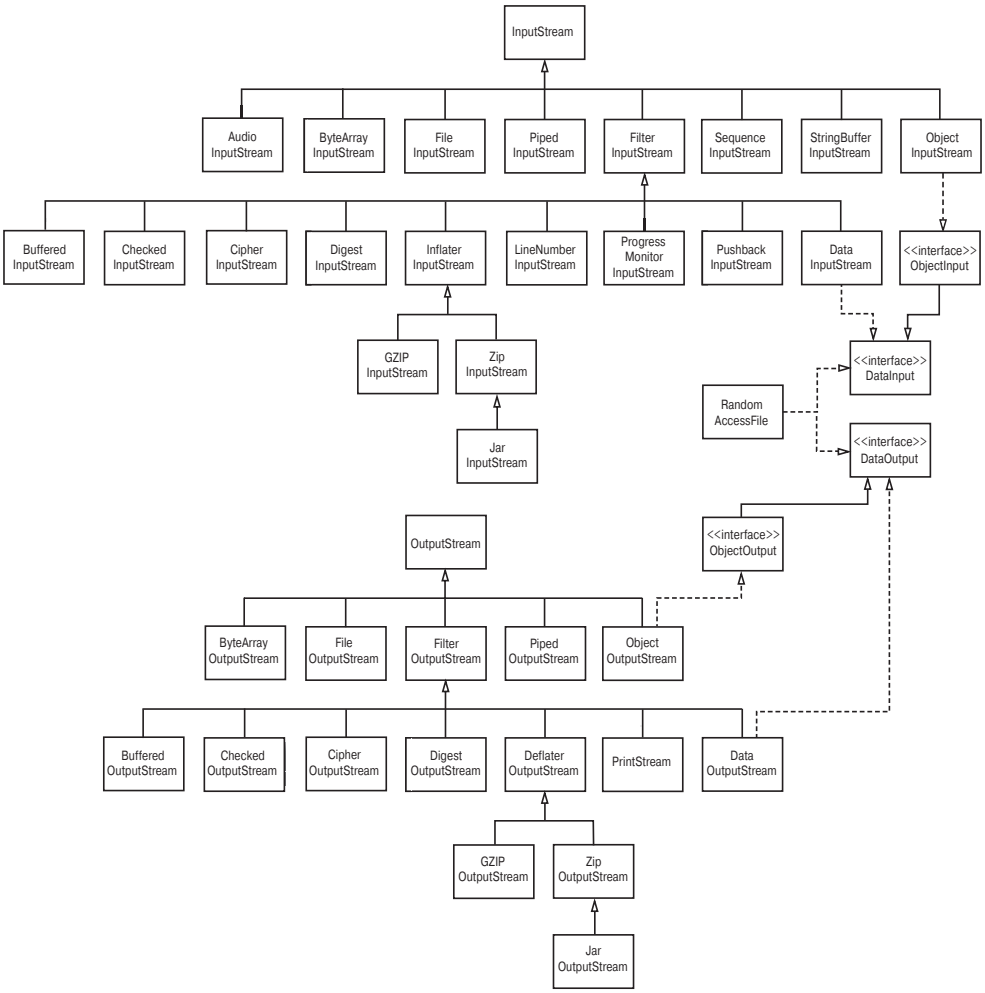


Рис. 1.1. Иерархия потоков входных и выходных данных

Как уже рассказывалось, классы `InputStream` и `OutputStream` позволяют выполнять считывание и запись отдельных байтов и массивов байтов. Эти классы образуют основу иерархии, показанной на рис. 1.1. Для выполнения считывания и записи строк и чисел требуются обладающие бо́льшими возможностями подклассы. Классы `DataInputStream` и `DataOutputStream`, например, позволяют выполнять считывание и запись всех простых типов данных Java в двоичном формате. И, наконец, есть потоки, которые просто позволяют выполнять определенные полезные операции, вроде потоков `ZipInputStream` и `ZipOutputStream`, которые позволяют считывать и записывать файлы в таком известном формате сжатия, как ZIP.

Для работы с текстом в формате Unicode, с другой стороны, необходимо применять подклассы таких абстрактных классов, как `Reader` и `Writer` (см. рис. 1.2). Базовые методы классов `Reader` и `Writer` похожи на базовые методы классов `InputStream` и `OutputStream`.

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

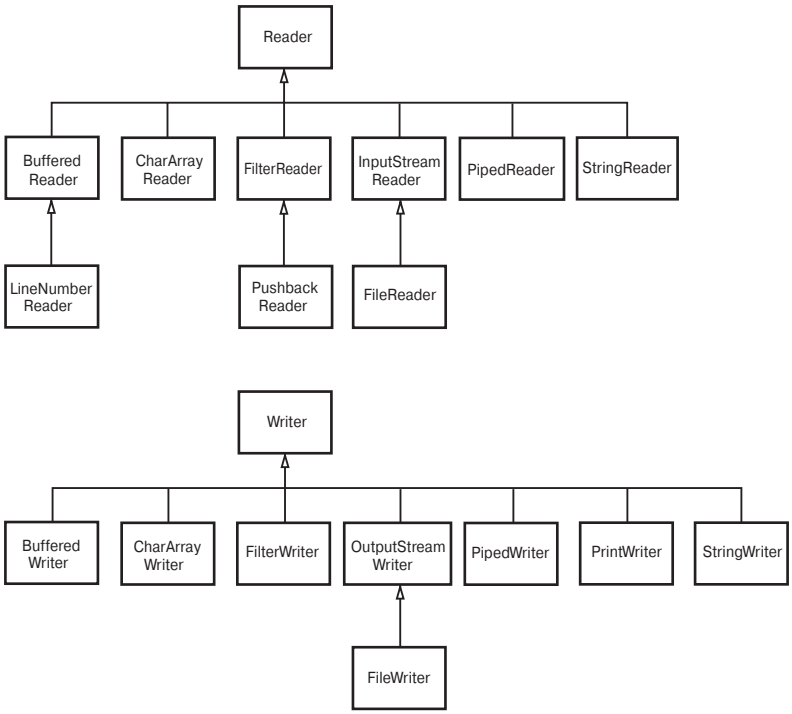


Рис. 1.2. Иерархия классов `Reader` *и* `Writer`

Метод `read` возвращает единицу кода Unicode (в виде целого числа от 0 до 65535) или `-1`, если был достигнут конец файла. А метод `write` вызывается с единицей кода Unicode. Более подробную информацию о том, что собой представляют единицы кода Unicode (Unicode code units), можно найти в главе 3 первого тома этой книги.

Версия Java SE 5.0 сделала доступными четыре дополнительных интерфейса: `Closeable`, `Flushable`, `Readable` и `Appendable` (рис. 1.3). Первые два из них являются очень простыми, и определяют, соответственно, метод

```
void close() throws IOException
```

и метод

```
void flush()
```

Классы `InputStream`, `OutputStream`, `Reader` и `Writer` реализуют интерфейс `Closeable`, а классы `OutputStream` и `Writer` — интерфейс `Flushable`.

Интерфейс `Readable` имеет один единственный метод

```
int read(CharBuffer cb)
```

Класс `CharBuffer` имеет методы для последовательного и произвольного доступа для чтения и записи. Он представляет либо буфер в памяти (in-memory), либо отображаемый в памяти файл. (Более подробно об этом будет рассказываться чуть позже в этой главе, в разделе “Структура буферных данных”.)

Интерфейс `Appendable` имеет два метода, которые позволяют присоединять как отдельные символы, так и целые последовательности символов:

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

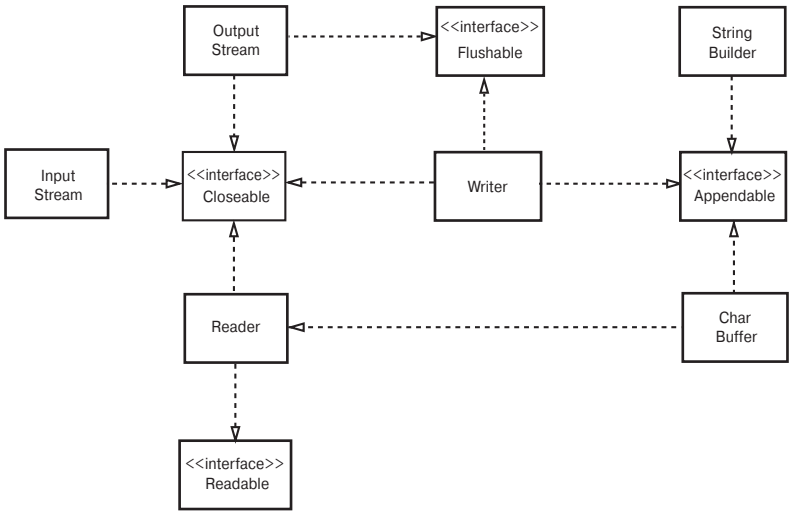


Рис. 1.3. Интерфейсы `Closeable`, `Flushable`, `Readable` *и* `Appendable`

Интерфейс `CharSequence` описывает базовые свойства последовательности значений `char`. Его реализуют такие классы, как `String`, `CharBuffer`, `StringBuilder` и `StringBuffer`.

Из всех представляющих потоки классов в коллекции только класс `Writer` реализует интерфейс `Appendable`.

API java.io.Closeable 5.0

- `void close()`

  Закрывает данный интерфейс `Closeable`. Этот метод может выдавать исключение `IOException`.

API java.io.Flushable 5.0

- `void flush()`

  Сбрасывает данный интерфейс `Flushable`.

API java.lang.Readable 5.0

- `int read(CharBuffer cb)`

  Пытается считать столько значений `char` в `cb`, сколько там может уместиться. Возвращает количество считанных значений или `-1`, если из данного интерфейса `Readable` больше не доступны никакие значения.

API java.lang.Appendable 5.0

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

  Присоединяет заданный блок кода или все блоки кода из заданной последовательности к данному интерфейсу `Appendable`; возвращает `this`.

API java.lang.CharSequence 1.4

- `char charAt(int index)`

  Возвращает единицу кода, соответствующую заданному индексу.

- `int length()`

  Возвращает информацию об общем количестве единиц кода в данной последовательности.

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

  Возвращает последовательность `CharSequence`, состоящую только из тех единиц кода, которые хранятся в промежутке, начиная с индекса `startIndex` и заканчивая индексом `endIndex - 1`.

- `String toString()`

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

Комбинирование потоковых фильтров

Классы `FileInputStream` и `FileOutputStream` позволяют создавать потоки входных и выходных данных и присоединять их к конкретному файлу на диске. Имя требуемого файла и полный путь к нему указываются в конструкторе.

Например, строка кода

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

будет приводить к поиску в каталоге пользователя файла с именем `employee.dat`.



Поскольку все классы в пакете `java.io` воспринимают относительные пути к файлам как команду начинать поиск с рабочего каталога пользователя, может потребоваться узнать, как именно выглядит этот каталог. Получить подобную информацию можно путем вызова метода `System.getProperty("user.dir")`.

Подобно абстрактным классам `InputStream` и `OutputStream`, классы `FileInputStream` и `FileOutputStream` поддерживают выполнение чтения и записи только на уровне байтов. То есть мы можем только считывать байты и массивы байтов из объекта `fin`.

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

Как читатель увидит в следующем разделе, если бы у нас только был класс `DataInputStream`, тогда мы могли бы считывать числовые типы:

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

Однако так же, как у класса `FileInputStream` нет методов для считывания числовых типов, у класса `DataInputStream` нет никакого метода для извлечения данных из файла.

В Java используется хитрый механизм для разделения двух видов обязанностей. Одни потоки (вроде `FileInputStream` и потока входных данных, возвращаемого методом `openStream` класса URL) могут извлекать байты из файлов и других, более экзотических мест, а другие потоки (вроде `DataInputStream` и `PrintWriter`) могут компоновать эти байты в более полезные типы данных. Использующему Java программисту нужно просто комбинировать их. Например, чтобы получить возможность счи-

тывать числа из файла, достаточно сначала создать поток FileInputStream, а затем передать его конструктору класса DataInputStream.

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

Если читатель снова посмотрит на рис. 1.1, то сможет увидеть там классы FilterInputStream и FilterOutputStream. Подклассы этих классов позволяют увеличивать возможности исходных потоков обработкой данных на уровне байтов.

За счет вложения фильтров можно добавлять очень много возможностей. Например, по умолчанию потоки не оснащаются буфером. То есть каждый вызов метода read приводит к запрашиванию у операционной системы отделения просто еще одного байта. Гораздо эффективнее вместо этого запрашивать сразу целые блоки данных и помещать их в буфер. Желание использовать буферизацию и методы ввода данных в файл требует применения следующей монстроподобной последовательности конструкторов:

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

Обратите внимание на то, что DataInputStream размещается *последним* в цепочке конструкторов, потому что планируется использовать методы DataInputStream и сделать так, чтобы *они* использовали оснащенный буфером метод read.

Иногда бывает необходимо отслеживать промежуточные потоки при соединении их в цепочку. Например, при считывании входных данных часто бывает необходимо заглядывать на следующий байт для получения уверенности в том, что в нем содержится именно то значение, которое ожидается. Для этой цели в Java предлагается класс PushbackInputStream.

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

Теперь теоретически можно считывать следующий байт:

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

и выдавать его обратно, если он является не тем, что нужно:

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

Но считывание (read) и отмена считывания (unread) являются *единственными* методами, которые можно применять к потоку PushbackInputStream. При желании заглядывать наперед, а также считывать числа, требуется уже применять не только поток PushBackInputStream, но и ссылку на поток DataInputStream.

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

Конечно, в библиотеках потоков других языков программирования такие полезные вещи, как буферизация и просмотр вперед (lookahead), обеспечиваются автоматически, так что в Java необходимость комбинировать для их получения фильтры потоков немного напрягает, но зато возможность смешивать и подбирать классы фильтров для создания по-настоящему полезных последовательностей потоков дает просто огромную гибкость. Например, путем использования следующей последовательности потоков можно считывать числа из сжатого ZIP-файла (рис. 1.4).

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

(Более подробно о том, как в Java можно работать с ZIP-файлами, будет рассказываться чуть позже в этой главе, в разделе “ZIP-архивы”.)

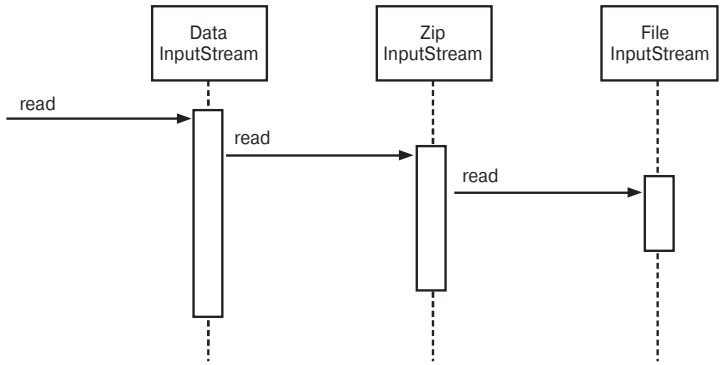


Рис. 1.4. Последовательность фильтруемых потоков

API java.io.FileInputStream 1.0

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

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

API java.io.FileOutputStream 1.0

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

  Создает новый поток выходных данных и помещает их в файл, который указывается либо в строке `name`, либо в объекте `file`. (О классе `File` более подробно будет рассказываться в конце этой главы.) Если параметр `append` равняется `true`, тогда данные добавляются в конец файла, и в случае обнаружения уже существующего файла с таким же именем, он не удаляется. В противном случае этот метод всегда удаляет любой обнаруженный существующий файл с таким же именем.

API java.io.BufferedInputStream 1.0

- `BufferedInputStream(InputStream in)`

  Создает буферизованный поток входных данных. Такой поток умеет выполнять считывание символов без постоянного обращения к устройству. Когда буфер пуст, в него считывается новый блок данных

API java.io.BufferedOutputStream 1.0

- `BufferedOutputStream(OutputStream out)`

  Создает буферизированный поток выходных данных. Такой поток умеет собирать подлежащие записи символы без постоянного обращения к устройству. Когда буфер полностью заполняется данными или выполняется очистка (flush) потока, данные записываются.

API java.io.PushbackInputStream 1.0

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

  Создает поток с однобайтовым буфером для просмотра вперед (lookahead buffer) или с буфером обратной выдачи (pushback buffer) указанного размера.

- `void unread(int b)`

  Выдает байт обратно, чтобы он мог быть извлечен снова при следующем запросе на считывание.

  Параметры: `b` Байт, который подлежит повторному считыванию.

Текстовые входные и выходные данные

При сохранении данных Java-программист может выбирать между двоичным и текстовым форматом. Например, в случае сохранения целого числа 1234 в двоичном формате оно будет записываться в виде такой последовательности байтов: `00 00 04 D2` (в шестнадцатеричном представлении), в текстовом же формате оно будет сохраняться просто в виде строки `"1234"`. Хотя в двоичном формате входные и выходные данные обрабатываются быстрее и эффективнее, они не очень удобны для восприятия человеком. Поэтому мы сначала расскажем о текстовых входных и выходных данных и только потом уже о двоичных.

При сохранении текстовых строк нужно продумывать *кодировку символов*. В случае использования кодировки UTF-16 строка `"1234"` будет кодироваться как `00 31 00 32 00 33 00 34` (в шестнадцатеричном представлении). Многие программы, однако, ожидают в текстовых файлах другой кодировки. В кодировке ISO 8859-1, которая наиболее часто применятся в США и Западной Европе, например, та же самая строка будет выглядеть уже следующим образом: `31 32 33 34` (т.е. не будет содержать нулевых байтов).

Класс `OutputStreamWriter` превращает поток символов Unicode в поток байтов, применяя выбранную кодировку символов, а класс `InputStreamReader` наоборот — превращает поток входных данных, содержащий байты (которые представляют символы в какой-нибудь кодировке) в считывателя, выдающего символы в кодировке Unicode.

Например, ниже показано, как можно создать считывателя входных данных (input reader), способного считывать набираемые на клавиатуре символы с консоли и преобразовывать их в Unicode:

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

Этот считыватель предполагает, что в системе используется стандартная кодировка символов, например, в Западной Европе это будет ISO 8859-1. Однако в принципе

можно выбирать и любую другую кодировку просто путем ее указания в конструкторе InputStreamReader, например:

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

Более подробно о кодировках символов будет рассказываться чуть позже в этой главе, в разделе "Наборы символов".

Поскольку считывателей (reader) и записывателей (writer) очень часто требуется присоединять к файлу, в Java специально для этой цели предлагается два таких удобных класса, как FileReader и FileWriter, Например, ниже показаны два абсолютно эквивалентных способа определения записывателя:

```
FileWriter out = new FileWriter("output.txt");
FileWriter out = new FileWriter(new FileOutputStream("output.txt"));
```

Запись текстовых выходных данных

Для текстовых выходных данных лучше всего использовать класс PrintWriter. Этот класс имеет методы для вывода строк и чисел в текстовом формате. Доступен даже удобный конструктор для связывания PrintWriter с FileWriter. Операторы

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

и

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

являются полностью эквивалентными.

Для выполнения записи в PrintWriter применяются все те же самые методы print, println и printf, что и в случае System.out. Эти методы можно использовать для вывода чисел (int, short, long, float, double), символов, значений boolean, строк и объектов.

Например, возьмем следующий код:

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

Он записывает символы

```
Harry Hacker 75000.0
```

в out. Эти символы затем преобразуются в байты и в конечном итоге оказываются в файле employee.txt.

Метод println добавляет к ним символ конца строки, являющийся корректным для целевой системы (для Windows это "\r\n", а для Unix — "\n"). А получается этот символ конца строки за счет вызова System.getProperty("line.separator").

Если для записывателя устанавливается *режим автоматической очистки буфера* (autoflush), тогда при каждом вызове метода println все содержащиеся в буфере в этот момент символы отправляются в их пункт назначения. (Классы PrintWriter всегда снабжаются буфером.) По умолчанию режим автоматической очистки *не* включается. Включать и отключать его можно путем использования конструктора PrintWriter(Writer out, boolean autoFlush):

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

Методы print не выдают исключений. Для проверки того, не случилось ли чего с потоком, можно вызывать метод checkError.



У давних почитателей Java может возникнуть вопрос о том, а что же случилось с классом `PrintStream` и с `System.out`? В версии Java 1.0 класс `PrintStream` просто усекал все символы Unicode до символов ASCII за счет отбрасывания старшего байта. Понятное дело, что такой подход не давал чистых и переносимых результатов, из-за чего и был усовершенствован добавлением в версии Java 1.1 классов типа `Reader` и `Writer`. Для обеспечения совместимости с существующим кодом `System.in`, `System.out` и `System.err` по-прежнему являются потоками, но не такими, которые могут считывать и записывать данные. Класс `PrintStream` теперь умеет внутренне преобразовывать символы Unicode в стандартную кодировку хоста тем же самым образом, что и класс `PrintWriter`. Объекты типа `PrintStream` работают точно так же, как и объекты типа `PrintWriter`, в случае использования методов `print` и `println`, но в отличие от них также позволяют и выводить необработанные байты с помощью методов `write(int)` и `write(byte[])`.



java.io.PrintWriter 1.1

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

  Создает новый экземпляр `PrintWriter`.

  Параметры: `out` Записыватель символьных (алфавитно-цифровых) выходных данных.

  `autoflush` При значении `true` методы `println` будут автоматически очищать буфер выходных данных (по умолчанию имеет значение `false`).
- `PrintWriter(OutputStream out)`
- `PrintWriter(OutputStream out, boolean autoflush)`

  Создает новый экземпляр `PrintWriter` из существующего экземпляра `OutputStream` путем создания необходимого промежуточного экземпляра `OutputStreamWriter`.
- `PrintWriter(String filename)`
- `PrintWriter(File file)`

  Создает новый экземпляр `PrintWriter`, который записывает данные в указанный файл путем создания необходимого промежуточного экземпляра `FileWriter`.
- `void print(Object obj)`

  Выводит объект путем отображения получившейся из `toString` строки.

  Параметры: `obj` Объект, который подлежит выводу.
- `void print(String s)`

  Выводит строку в формате Unicode.
- `void println(String s)`

  Выводит строку вместе с символом окончания строки. Очищает поток, если поток работает в режиме автоматической очистки (autoflush).
- `void print(char[] s)`

  Выводит все имеющиеся в указанном массиве символы в формате Unicode.

- `void print(char c)`

  Выводит символ в формате Unicode.
- `void print(int i)`
- `void print(long l)`
- `void print(float f)`
- `void print(double d)`
- `void print(boolean b)`

  Выводит заданное значение в текстовом формате.
- `void printf(String format, Object... args)`

  Выводит заданные значения в соответствии с требованиями, указанными в строке форматирования. О том, как задается строка форматирования, рассказывалось в главе 3 первого тома этой книги.
- `boolean checkError()`

  Возвращает `true`, если возникла ошибка с форматированием или выходными данными. В случае возникновения ошибки поток считается запорченным, и тогда все вызовы метода `checkError` возвращают `true`.

Считывание текстовых входных данных

Уже известно, что:

- для записи данных в двоичном формате применяется класс `DataOutputStream`;
- для записи данных в текстовом формате используется класс `PrintWriter`.

Поэтому можно и догадаться, что у класса `DataInputStream` существует аналог, позволяющий считывать данные в текстовом формате. Наиболее близким аналогом является класс `Scanner`, способы применения которого довольно обширно демонстрировались в первом томе этой книги. До выхода версии Java SE 5.0, однако, единственным возможным вариантом для обработки входных данных в текстовом формате был класс `BufferedReader`. Этот класс имеет метод `readLine`, который позволяет выполнять считывание строки текста, и может применяться только в сочетании с источником данных:

```
BufferedReader in = new BufferedReader(new FileReader("employee.txt"));
```

Когда никаких входных данных больше не доступно, метод `readLine` возвращает `null`. Типичный цикл обработки входных данных, следовательно, выглядит следующим образом:

```
String line;
while ((line = in.readLine()) != null)
{
  Выполнение каких-то операций с line
}
```

Однако у класса `BufferedReader` нет никаких методов для считывания чисел. Поэтому мы рекомендуем все-таки применять для считывания входных данных в текстовом формате не этот класс, а `Scanner`.

Сохранение объектов в текстовом формате

В этом разделе для примера рассматривается программа, сохраняющая массив записей `Employee` в текстовом файле. Каждая запись сохраняется в отдельной строке.

Поля экземпляра отделяются друг от друга с помощью разделителя. В качестве этого разделителя здесь применяется символ вертикальной черты (|). (Другим популярным кандидатом на эту роль является символ двоеточия (:). Однако в принципе каждый разработчик может использовать свой разделитель.) Естественно, вопрос о том, что может произойти в случае, если символ | встретится непосредственно в одной из сохраняемых строк, пока не затрагивается.

Ниже приведен демонстрационный набор записей:

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

Процесс записи выглядит очень просто. Поскольку запись выполняется в текстовый файл, применяется класс PrintWriter. Мы просто записываем все поля, завершая их либо символом |, либо, если речь идет уже о последнем поле, комбинацией символов \n. Эта работа выполняется в рамках следующего метода writeData, который мы добавляем к нашему классу 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 мы делаем это по одной строке за раз и разделяем поля. Мы используем сканер (Scanner) для считывания каждой строки и затем разбиваем строку на лексемы (tokens) с помощью метода String.split.

```
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 роли параметра для метода split выступает регулярное выражение, описывающее разделитель (separator). О регулярных выражениях более подробно будет рассказываться в конце этой главы. Так сложилось, что символ вертикальной черты (|) имеет в регулярных выражениях специальное значение, поэтому он должен обязательно отменяться с помощью символа \, а тот, в свою очередь, тоже должен отменяться с помощью еще одного символа \, что в результате приводит к получению такого выражения: \\|.

Весь код данной демонстрационной программы приведен в листинге 1.1.

Статический метод

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

сначала записывает длину массива, а затем — каждую запись.

Статический метод

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

сначала считывает длину массива, а затем уже считывает каждую запись. Это оказывается не совсем простым процессом:

```
int n = in.nextInt();
in.nextLine(); // "расходование" символа 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 замыкающего символа новой строки. Этот символ должен обязательно расходоваться, чтобы метод `readData` мог переходить к следующей строке входных данных при вызове `nextLine`.

Листинг 1.1. Содержимое файла `TextFileTest.java`

```
1.  import java.io.*;
2.  import java.util.*;
3.
4.  /**
5.   * @version 1.12 2007-06-22
6.   * @author Cay Horstmann
7.   */
8.  public class TextFileTest
9.  {
10.   public static void main(String[] args)
11.   {
12.     Employee[] staff = new Employee[3];
13.
14.     staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
15.     staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
16.     staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
17.
18.   try
19.   {
20.     // сохранение всех записей о сотрудниках в файле employee.dat
21.     PrintWriter out = new PrintWriter("employee.dat");
22.     writeData(staff, out);
23.     out.close();
24.     // извлечение всех записей в новый массив
25.     Scanner in = new Scanner(new FileReader("employee.dat"));
26.     Employee[] newStaff = readData(in);
27.     in.close();
28.    // вывод новых считанных записей о сотрудниках
29.     for (Employee e : newStaff)
30.       System.out.println(e);
31.   }
32.   catch (IOException exception)
33.   {
34.     exception.printStackTrace();
35.   }
36. }
```

```
37.
38.  /**
39.   * Записывает все данные о сотрудниках в массиве
40.   * в класс PrintWriter
41.   * @param employees массив данных о сотрудниках
42.   * @param out класс PrintWriter
43.  */
44.  private static void writeData(Employee[] employees, PrintWriter out)
45.     throws IOException
46.  {
47.    // запись количества сотрудников
48.    out.println(employees.length);
49.
50.    for (Employee e : employees)
51.    e.writeData(out);
52.  }
53.
54.  /**
55.   * Считывает массив данных о сотрудниках из класса Scanner
56.   * @param in класс Scanner
57.   * @return массив данных о сотрудниках
58.  */
59.  private static Employee[] readData(Scanner in)
60.  {
61.    // извлечение размера массива
62.    int n = in.nextInt();
63.    in.nextLine(); // расходование символа новой строки
64.
65.    Employee[] employees = new Employee[n];
66.    for (int i = 0; i < n; i++)
67.    {
68.      employees[i] = new Employee();
69.      employees[i].readData(in);
70.    }
71.      return employees;
72.    }
73.  }
74.
75.  class Employee
76.  {
77.    public Employee()
78.    {
79.    }
80.
81.    public Employee(String n, double s, int year, int month, int day)
82.    {
83.      name = n;
84.      salary = s;
85.      GregorianCalendar calendar = new GregorianCalendar(year, month - 1,  day);
86.      hireDay = calendar.getTime();
87.    }
88.
89.    public String getName()
90.    {
91.      return name;
92.    }
93.
```

```
94.   public double getSalary()
95.   {
96.     return salary;
97.   }
98.
99.   public Date getHireDay()
100.  {
101.    return hireDay;
102.  }
103.
104.  public void raiseSalary(double byPercent)
105.  {
106.    double raise = salary * byPercent / 100;
107.    salary += raise;
108.  }
109.
110.  public String toString()
111.  {
112.    return getClass().getName() + "[name=" + name + ",salary=" +
113.       salary + ",hireDay=" + hireDay + "]";
114.  }
115.
116. /**
117.  * Записывает данные о сотрудниках
118.  * в класс PrintWriter
119.  * @param out класс PrintWriter
120.  */
121. public void writeData(PrintWriter out)
122.  {
123.    GregorianCalendar calendar = new GregorianCalendar();
124.    calendar.setTime(hireDay);
125.    out.println(name + "|" + salary + "|" + calendar.get(Calendar.YEAR) +
126.      "|" + (calendar.get(Calendar.MONTH) + 1) + "|"
127.       + calendar.get(Calendar.DAY_OF_MONTH));
128.  }
129.
130. /**
131.  * Считывает данные о сотрудниках из буферизованного класса Scanner
132.  * @param in класс Scanner
133.  */
134. public void readData(Scanner in)
135. {
136.   String line = in.nextLine();
137.   String[] tokens = line.split("\\|");
138.   name = tokens[0];
139.   salary = Double.parseDouble(tokens[1]);
140.   int y = Integer.parseInt(tokens[2]);
141.   int m = Integer.parseInt(tokens[3]);
142.   int d = Integer.parseInt(tokens[4]);
143.   GregorianCalendar calendar = new GregorianCalendar(y, m - 1, d);
144.   hireDay = calendar.getTime();
145. }
146.
147.  private String name;
148.  private double salary;
149.  private Date hireDay;
150. }
```

Наборы символов

Раньше во всей библиотеке Java не было никакого систематического способа для работы с наборами символов. Пакет `java.nio`, появившийся еще в версии Java SE 1.4, изменил эту ситуацию, сделав доступным класс `Charset`. (Обратите внимание на то, что `s` в имени этого класса является строчной.)

Каждый набор символов отображается между последовательностями двухбайтовых единиц кода Unicode и байтовыми последовательностями, используемыми в локальной кодировке символов. Одной из наиболее популярных среди кодировок является ISO-8859-1, которая представляет собой однобайтовую кодировку первых 256 символов Unicode. Все большую значимость еще набирает и такая кодировка, как ISO-8859-15, которая заменяет некоторые менее удобные символы из кодировки ISO-8859-1 с акцентированными буквами, используемыми во французском и финском языках, и, что даже еще более важно, замещает символ "международной валюты" (¤) символом евро (€) в точке кода `0xA4`. Другими примерами кодировок символов являются кодировки с переменным количеством байтов, наиболее часто применяемые для японского и китайского языков.

Класс `Charset` предполагает использование только тех имен наборов символов, которые были стандартизированы комитетом IANA (Internet Assigned Numbers [Names] Authority — комитет по присвоению номеров (имен) Internet) и перечислены в его реестре наборов символов (`http://www.iana.org/assignments/character-sets`). Эти имена немного отличаются от тех, что применялись в предыдущих версиях. Например, "официальное" имя набора ISO-8859-1 теперь выглядит как `"ISO-8859-1"`, а не так `"ISO8859_1"`, как оно выглядело вплоть до версии Java SE 1.3.



Прекрасный источник информации по ISO 8859 доступен по адресу: `http://czyborra.com/charsets/iso8859.html`.

Получается доступ к `Charset` путем вызова статического метода `forName` с либо официальными именем, либо с одним из соответствующих псевдонимов:

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

Имена наборов символов не чувствительны к регистру.

Для совместимости с другими соглашениями об именовании каждый набор символов может иметь ряд псевдонимов. Например, набор ISO-8859-1 имеет такие 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 `Set` с псевдонимами. Код, необходимый для осуществления прохода по псевдонимам, выглядит следующим образом:

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

Для выяснения того, какие наборы символов доступны в конкретной реализации, нужно вызывать метод `availableCharsets`. Код, необходимый 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 обязательно должны присутствовать в каждой реализации Java, в табл. 1.2 — схемы кодировок, которые устанавливаются вместе с JDK (Java Development Kit) по умолчанию, а в табл. 1.3 — наборы символов, которые устанавливаются только на тех операционных системах, где используются не европейские 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-битный формат преобразования символов Unicode.
UTF-16	UTF-16	16-битный формат преобразования символов Unicode, порядок байтов задается с помощью необязательной начальной отметки порядка байтов.
UTF-16BE	UnicodeBigUnmarked	16-битный формат преобразования символов Unicode с порядком следования байтов начиная со старшего.
UTF-16LE	UnicodeLittleUnmarked	16-битный формат преобразования символов Unicode с порядком следования байтов начиная с младшего.

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).

Окончание табл. 1.2

Стандартное название кодировки	Прежнее название	Описание
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, стандарт 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).

Получив набор символов, можно начинать использовать его для преобразования строк Unicode в закодированные последовательности байтов. Необходимый для кодирования строки Unicode код выглядит следующим образом:

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

Для выполнения обратной операции, т.е. для декодирования последовательности байтов, необходим байтовый буфер. Получить его можно из байтового массива с использованием статического метода `wrap` массива `ByteBuffer`. Метод `decode` в результате выдает объект `CharBuffer`. Вызов для него метода `toString` позволяет получить уже строку.

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

API java.nio.charset.Charset 1.4

- `static SortedMap availableCharsets()`
  Получает все наборы символов, доступные для данной виртуальной машины. Возвращает таблицу (map), в роли ключей в которой выступают имена наборов символов, а в роли значений — сами наборы символов.
- `static Charset forName(String name)`
  Получает набор символов для указанного имени.
- `Set aliases()`
  Возвращает набор псевдонимов для данного набора символов.
- `ByteBuffer encode(String str)`
  Кодирует указанную строку в последовательность байтов.
- `CharBuffer decode(ByteBuffer buffer)`
  Декодирует указанную последовательность байтов. Нераспознанные входные символы преобразуются в Unicode-символы замены (`'\uFFFD'`).

API 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 указанным массивом байтов или диапазоном.

API java.nio.CharBuffer

- `char[] array()`
  Возвращает массив блоков кода, за управление которыми отвечает данный буфер.
- `char charAt(int index)`
  Возвращает блок кода, находящийся по указанному индексу.
- `String toString()`
  Возвращает строку, состоящую из блоков кода, за управление которыми отвечает данный буфер.

Чтение и запись двоичных данных

Интерфейс `DataOutput` определяет следующие методы для записи чисел, символов, значений `Boolean` или строк в двоичном формате:

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

Например, метод `writeInt` всегда записывает целое число в виде 4-байтовой двоичной величины независимо от количества цифр, а метод `writeDouble` всегда записывает значение `double` в виде 8-байтовой двоичной величины. Результирующий вывод не понятен для человека, но зато объем требуемого пространства будет одинаковым для каждого значения заданного типа, и обратное считывание таких значений будет осуществляться гораздо быстрее, чем синтаксический анализ текста.



Для сохранения целых чисел и чисел с плавающей запятой существует два разных способа, зависящих от используемой платформы. Допустим, например, что имеется некое 4-байтовое значение типа `int`, скажем, десятичное число `1234`, или `4D2` в шестнадцатеричном формате `(1234 = 4 × 256 + 13 × 16 + 2)`. Оно может быть сохранено таким образом, чтобы первых четыре байта в памяти занимал его самый старший байт (Most Significant Byte — MSB): `00 00 04 D2`. Такой подход называется сохранением в формате с порядком следования байтов начиная со старшего (big-endian). Или же оно может быть сохранено так, чтобы первым шел, наоборот, самый младший байт: `D2 04 00 00`. Такой подход, соответственно, называется сохранением в формате с порядком следования байтов, начиная с младшего (little-endian). На платформах SPARC, например, применяется формат с порядком следования байтов, начиная со старшего, а на платформах Pentium — формат с порядком следования байтов, начиная с младшего. Это может приводить к возникновению проблем. При сохранении файла, написанного на C или C++, данные сохраняются в точности тем образом, которым их сохраняет процессор. Этот факт делает перемещение даже простейших файлов данных с одной платформы на другую довольно непростой задачей. В Java же все значения записываются в формате с порядком следования байтов, начиная со старшего, независимо от процессора, что, соответственно, делает файлы данных Java не зависящими от используемой платформы.

Метод `writeUTF` записывает строковые данные с применением модифицированной версии 8-битного формата преобразования Unicode (8-bit Unicode Transformation Format). Вместо того чтобы просто использовать стандартную кодировку UTF-8 (табл. 1.4), символьные строки сначала представляются в кодировке UTF-16 (табл. 1.5), после чего полученный результат уже кодируется с использованием правил UTF-8. Для символов с кодом выше `0xFFFF` модифицированная кодировка выглядит по-другому. Она применяется для обеспечения обратной совместимости с теми виртуальными машинами, которые были созданы во времена, когда Unicode еще не могла выходить за рамки 16 битов.

Поскольку такой модифицированной версией UTF-8 больше никто не пользуется, для записи строк, предназначенных для виртуальной машины Java, например, в случае создания программы, генерирующей байт-код, следует использовать только метод `writeUTF`, а для всех остальных целей нужно применять 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 $b_{19}b_{18}b_{17}b_{16} = a_{20}a_{19}a_{18}a_{17}a_{16} - 1$



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 считывания данных обратно можно использовать следующие методы, определенные в интерфейсе `DataInput`:

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

Реализуется интерфейс `DataInput` классом `DataInputStream`. Считывать двоичные данные из файла можно просто путем комбинирования `DataInputStream` с нужным источником байтов, например, `FileInputStream`:

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

Подобным образом записывать двоичные данные можно с помощью класса `DataOutputStream`, который реализует интерфейс `DataOutput`:

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

API `java.io.DataInput 1.0`

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

  Считывает значение указанного типа.

- `void readFully(byte[] b)`
  Считывает байты в массив `b`, устанавливая блокировку до тех пор, пока не будут считаны все байты.
  Параметры: `b` Буфер, в который считываются данные.
- `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` Буфер, в который считываются данные.
  `off` Начальное смещение данных.
  `len` Максимальное количество подлежащих считыванию байтов.
- `String readUTF()`
  Считывает строку символов 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.

API **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)`
  Записывает значение указанного типа.
- `void writeChars(String s)`
  Записывает все символы из строки.
- `void writeUTF(String s)`
  Записывает строку символов в "модифицированном формате UTF-8".

Файлы с с произвольным доступом

Класс `RandomAccessFile` позволяет отыскивать или записывать данные в любом месте в файле. Файлы на дисках всегда имеют возможность произвольного доступа, а вот потоки данных, поступающие из сети — нет. Файл с произвольным доступом может открываться как только для чтения, так и для чтения и записи одновременно. Требуемый режим задается указанием во втором параметре конструктора либо строки `"r"` (режим "только для чтения"), либо строки `"rw"` (режим "для чтения и записи") соответственно:

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

В случае открытия существующего файла как `RandomAccessFile`, он не удаляется.

У любого файла с произвольным доступом имеется так называемый *указатель файла* (file pointer), указывающий на позицию следующего байта, который будет считываться или записываться. Метод `seek` устанавливает этот указатель на произвольную байтовую позицию внутри файла. Этому методу в качестве аргумента может передаваться целое число `long` в диапазоне от нуля до числа, представляющего всю длину файла в байтах.

Метод `getFilePointer` возвращает текущую позицию указателя файла.

Класс `RandomAccessFile` реализует как интерфейс `DataInput`, так и интерфейс `DataOutput`. Для считывания данных из и записи данных в файл с произвольной выборкой применяются методы вроде `readInt/writeInt` и `readChar/writeChar`, о которых рассказывалось в предыдущем разделе.

Теперь давайте для примера рассмотрим программу, сохраняющую записи о сотрудниках в файле с произвольным доступом. Все записи будут иметь одинаковые размеры. Это упрощает процесс считывания произвольной записи. Предположим, что указатель файла требуется поместить на третью запись. Тогда достаточно просто установить указатель файла на соответствующей байтовой позиции и можно приступать к считыванию.

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

При желании изменить запись и затем сохранить ее снова в том же месте, нужно обязательно не забыть вернуть указатель файла обратно в начало записи:

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

Для определения общего количества байтов в файле используется метод `length`. Общее количество записей можно вычислить путем деления длины файла на размер каждой записи.

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

Целые числа и значения с плавающей запятой имеют в двоичном формате фиксированный размер, а вот строки требуют приложения кое-каких дополнительных усилий. Для выполнения записи и считывания строк фиксированного размера мы добавляем два вспомогательных метода.

Метод `writeFixedString` записывает указанное количество единиц кода, начиная от начала строки. (Если единиц кода слишком мало, он дополняет строку 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);
  }
}
```

Метод `readFixedString` считывает символы из потока входных данных (`InputStream`) до тех пор, пока либо не прочитает указанное в `size` количество единиц кода, либо не встретит символ с нулевым значением, в случае чего он пропус-

кает все остальные нулевые значение в поле ввода. Для большей эффективности он использует для считывания строки 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();
}
```

Мы поместили методы writeFixedString и readFixedString внутрь вспомогательного класса DataIO.

Для сохранения записи фиксированного размера мы просто записываем все поля в двоичном формате.

```
public void writeData(DataOutput out) throws IOException
{
  DataIO.writeFixedString(name, NAME_SIZE, out);
  out.writeDouble(salary);
  GregorianCalendar calendar = new GregorianCalendar();
  calendar.setTime(hireDay);
  out.writeInt(calendar.get(Calendar.YEAR));
  out.writeInt(calendar.get(Calendar.MONTH) + 1);
  out.writeInt(calendar.get(Calendar.DAY_OF_MONTH));
}
```

Обратное считывание данных выполняется так же просто.

```
public void readData(DataInput in) throws IOException
{
  name = DataIO.readFixedString(NAME_SIZE, in);
  salary = in.readDouble();
  int y = in.readInt();
  int m = in.readInt();
  int d = in.readInt();
  GregorianCalendar calendar = new GregorianCalendar(y, m - 1, d);
  hireDay = calendar.getTime();
}
```

Давайте подсчитаем размер каждой записи. Для строк имен мы будем использовать по 40 символов. Следовательно, каждая запись будет содержать по 100 байтов:

- по 40 символов, т.е. по 80 байтов, для имени;
- по 1 числу типа double, т.е. по 8 байтов, для зарплаты;
- по 3 числа типа int, т.е. по 12 байтов, для даты.

Программа, код которой приведен в листинге 1.2, записывает три записи в файл данных, а затем считывает их оттуда в обратном порядке. Для эффективного осуществления этого процесса требуется возможность выполнения произвольного доступа, а именно — возможность получения доступа сначала к третьей записи.

Листинг 1.2. Содержимое файла `RandomFileTest.java`

```
1.  import java.io.*;
2.  import java.util.*;
3.
4.  /**
5.    * @version 1.11 2004-05-11
6.    * @author Cay Horstmann
7.    */
8.
9.  public class RandomFileTest
10. {
11.   public static void main(String[] args)
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
20.     {
21.       // сохранение всех записей о сотрудниках в файле employee.dat
22.       DataOutputStream out = new DataOutputStream(new FileOutputStream("employee.dat"));
23.       for (Employee e : staff)
24.         e.writeData(out);
25.       out.close();
26.
27.     // извлечение всех записей в новый массив
28.     RandomAccessFile in = new RandomAccessFile("employee.dat", "r");
29.     // вычисление рамзера массива
30.     int n = (int)(in.length() / Employee.RECORD_SIZE);
31.     Employee[] newStaff = new Employee[n];
32.
33.     // считывание записей о сотрудниках в обратном порядке
34.     for (int i = n - 1; i >= 0; i--)
35.     {
36.       newStaff[i] = new Employee();
37.       in.seek(i * Employee.RECORD_SIZE);
38.         newStaff[i].readData(in);
39.       }
40.       in.close();
41.
42.       // вывод считанных записей о сотрудниках
43.       for (Employee e : newStaff)
44.         System.out.println(e);
45.     }
46.     catch (IOException e)
47.     {
48.       e.printStackTrace();
49.     }
50.   }
51. }
52.
53. class Employee
54. {
55.   public Employee() {}
56.
```

```
57.   public Employee(String n, double s, int year, int month, int day)
58.   {
59.     name = n;
60.     salary = s;
61.     GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
62.    hireDay = calendar.getTime();
63.   }
64.
65.   public String getName()
66.   {
67.     return name;
68.   }
69.   public double getSalary()
70.   {
71.     return salary;
72.   }
73.
74.   public Date getHireDay()
75.   {
76.     return hireDay;
77.   }
78.
79.  /**
80.     Поднятие зарплаты данного сотрудника.
81.     @byPercent процент поднятия
82.  */
83.  public void raiseSalary(double byPercent)
84.  {
85.    double raise = salary * byPercent / 100;
86.    salary += raise;
87.  }
88.
89.  public String toString()
90.  {
91.    return getClass().getName()
92.      + "[name=" + name
93.    + ",salary=" + salary
94.    + ",hireDay=" + hireDay
95.    + "]";
96.  }
97.
98.    /**
99.      Запись информации о сотруднике в выходные данные
100.     @param out выходные данные
101.   */
102.   public void writeData(DataOutput out) throws IOException
103.   {
104.     DataIO.writeFixedString(name, NAME_SIZE, out);
105.     out.writeDouble(salary);
106.
107.     GregorianCalendar calendar = new GregorianCalendar();
108.     calendar.setTime(hireDay);
109.     out.writeInt(calendar.get(Calendar.YEAR));
110.     out.writeInt(calendar.get(Calendar.MONTH) + 1);
111.     out.writeInt(calendar.get(Calendar.DAY_OF_MONTH));
112.   }
113.
```

```
114.  /**
115.    Считывание данных сотрудника из входных данных
116.    @param in входные данные
117.  */
118.  public void readData(DataInput in) throws IOException
119.  {
120.    name = DataIO.readFixedString(NAME_SIZE, in);
121.    salary = in.readDouble();
122.    int y = in.readInt();
123.    int m = in.readInt();
124.    int d = in.readInt();
125.    GregorianCalendar calendar = new GregorianCalendar(y, m - 1, d);
126.    hireDay = calendar.getTime();
127.  }
128.
129.  public static final int NAME_SIZE = 40;
130.  public static final int RECORD_SIZE = 2 * NAME_SIZE + 8 + 4 + 4 + 4;
131.
132.  private String name;
133.  private double salary;
134.  private Date hireDay;
135. }
136.
137. class DataIO
138. {
139.  public static String readFixedString(int size, DataInput in)
140.    throws IOException
141.  {
142.    StringBuilder b = new StringBuilder(size);
143.    int i = 0;
144.    boolean more = true;
145.    while (more && i < size)
146.    {
147.      char ch = in.readChar();
148.      i++;
149.      if (ch == 0) more = false;
150.      else b.append(ch);
151.    }
152.    in.skipBytes(2 * (size - i));
153.    return b.toString();
154.  }
155.
156.  public static void writeFixedString(String s, int size, DataOutput out)
157.    throws IOException
158.  {
159.    for (int i = 0; i < size; i++)
160.    {
161.      char ch = 0;
162.      if (i < s.length()) ch = s.charAt(i);
163.      out.writeChar(ch);
164.    }
165.  }
166. }
```

API java.io.RandomAccessFile 1.0

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

  Параметры: `file` Файл для открытия.

  `mode` `r` — для режима "только для чтения", `rw` — для режима чтения и записи, `rws` — для режима чтения и записи с синхронным записыванием на диск данных и метаданных при каждом обновлении и `rwd` — для режима чтения и записи с синхронным записыванием на диск только данных.

- `long getFilePointer()`

  Возвращает информацию о текущем местонахождении файлового указателя.

- `void seek(long pos)`

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

- `long length()`

  Возвращает длину файла в байтах.

ZIP-архивы

ZIP-архивы позволяют хранить один и более файлов в (обычно) сжатом формате. У каждого ZIP-архива имеется заголовок, содержащий информацию вроде имени файла или использовавшегося для него метода сжатия. В Java для чтения ZIP-архивов применяется класс `ZipInputStream`. В каждом таком архиве всегда требуется просматривать отдельные *записи* (entries). Метод `getNextEntry` возвращает описывающий запись объект типа `ZipEntry`. Метод `read` класса `ZipInputStream` изменяется так, чтобы он возвращал `-1` в конце текущий записи (а не просто в конце ZIP-файла). Далее вызывается метод `closeEntry` для получения возможности перехода к считыванию следующей записи. Ниже приведена типичная кодовая последовательность для выполнения считывания содержимого ZIP-файла:

```
ZipInputStream zin = new ZipInputStream(new FileInputStream(zipname));
ZipEntry entry;
while ((entry = zin.getNextEntry()) != null)
{
  анализ entry;
  считывание содержимого zin;
  zin.closeEntry();
}
zin.close();
```

Для считывания содержимого конкретной записи из ZIP-файла эффективнее использовать не стандартный метод `read`, а методы какого-то обладающего бо́льшими возможностями потокового фильтра. Например, для считывания текстового файла, находящегося внутри ZIP-архива, можно применить следующий цикл:

```
Scanner in = new Scanner(zin);
while (in.hasNextLine())
  выполнение каких-то операций с in.nextLine();
```



В случае возникновения ошибки при считывании ZIP-файла класс `ZipInputStream` выдает исключение `ZipException`. Обычно подобное происходит при повреждении ZIP-файла.

Для записи ZIP-файла применяется класс `ZipOutputStream`. Для каждой записи, которую требуется поместить в ZIP-файл, создается объект `ZipEntry`. Желаемое имя для файла передается конструктору `ZipEntry`; тот устанавливает остальные параметры, вроде даты создания файла и метода распаковки. При желании эти параметры могут переопределяться. Далее вызывается метод `putNextEntry` класса `ZipOutputStream` для начала процесса записи нового файла. После этого данные самого файла отправляются потоку ZIP. По завершении вызывается метод `closeEntry`. Затем все эти действия выполняются повторно для всех остальных файлов, которые требуется сохранить в ZIP-архиве. Ниже приведена общая схема необходимого кода:

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



JAR-файлы (о которых рассказывалось в главе 10 первого тома этой книги) представляют собой те же ZIP-файлы, но только содержат записи несколько иного вида, называемые манифестом. Для считывания и записи манифестов применяются классы `JarInputStream` и `JarOutputStream`.

ZIP-потоки являются прекрасным примером мощи потоковой абстракции. При считывании данных, которые хранятся в сжатом виде, не нужно волноваться о том, будут ли они распаковываться по мере запрашивания. И источником байтов в ZIP-форматах вовсе не обязательно должен быть именно файл: ZIP-данные могут поступать и через сетевое подключение. На самом деле, при всяком считывании JAR-файла загрузчик классов аплета будет считывать и распаковывать данные именно из сети.

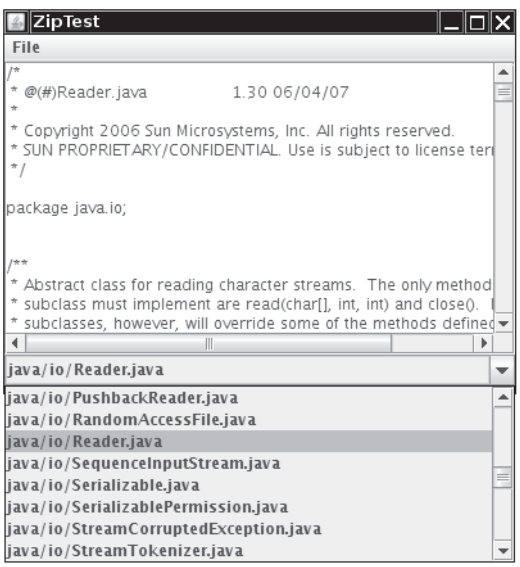


Рис. 1.5. Программа `ZipTest`



О том, как изменять ZIP-архив, можно прочитать в статье, доступной по адресу `http://www.javaworld.com/javaworld/jw-10-2000/jw-1027-toolbox.html`.

Программа, код которой приведен в листинге 1.3, позволяет открывать ZIP-архив, после чего отображает все хранящиеся внутри него файлы в комбинированном списке в нижней части экрана. В случае выбора какого-то одного из этих файлов она отображает его содержимое в текстовой области, как показано на рис. 1.5.

Листинг 1.3. Содержимое файла `ZipTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.io.*;
4.  import java.util.*;
5.  import java.util.List;
6.  import java.util.zip.*;
7.  import javax.swing.*;
8.  /**
9.   * @version 1.32 2007-06-22
10.  * @author Cay Horstmann
11.  */
12. public class ZipTest
13. {
14.   public static void main(String[] args)
15.   {
16.     EventQueue.invokeLater(new Runnable()
17.       {
18.         public void run()
19.         {
20.          ZipTestFrame frame = new ZipTestFrame();
21.          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22.          frame.setVisible(true);
23.         }
24.       });
25.     }
26.   }
27.  /**
28.   * Фрейм с текстовой областью для отображения содержимого файла из
29.   * ZIP-архива, комбинированным списком для выбора различных файлов из
30.   * архива и меню для загрузки нового архива.
31.   */
32.   class ZipTestFrame extends JFrame
33.   {
34.     public ZipTestFrame()
35.     {
36.       setTitle("ZipTest");
37.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
38.       // добавление меню и пунктов меню Open (Открыть) и Exit (Выход)
39.       JMenuBar menuBar = new JMenuBar();
40.       JMenu menu = new JMenu("File");
41.       JMenuItem openItem = new JMenuItem("Open");
42.       menu.add(openItem);
43.       openItem.addActionListener(new ActionListener()
44.       {
45.         public void actionPerformed(ActionEvent event)
46.         {
47.          JFileChooser chooser = new JFileChooser();
48.          chooser.setCurrentDirectory(new File("."));
49.          int r = chooser.showOpenDialog(ZipTestFrame.this);
50.          if (r == JFileChooser.APPROVE_OPTION)
51.          {
52.            zipname = chooser.getSelectedFile().getPath();
53.            fileCombo.removeAllItems();
54.            scanZipFile();
55.          }
56.         }
57.       });
```

```
58.    JMenuItem exitItem = new JMenuItem("Exit");
59.    menu.add(exitItem);
60.    exitItem.addActionListener(new ActionListener()
61.      {
62.        public void actionPerformed(ActionEvent event)
63.        {
64.           System.exit(0);
65.        }
66.     });
67.
68.    menuBar.add(menu);
69.    setJMenuBar(menuBar);

70.    // добавление текстовой области и комбинированного списка
71.    fileText = new JTextArea();
72.    fileCombo = new JComboBox();
73.    fileCombo.addActionListener(new ActionListener()
74.      {
75.        public void actionPerformed(ActionEvent event)
76.        {
77.         loadZipFile((String) fileCombo.getSelectedItem());
78.        }
79.      });
80.
81.    add(fileCombo, BorderLayout.SOUTH);
82.    add(new JScrollPane(fileText), BorderLayout.CENTER);
83.  }
84.
85.  /**
86.    * Сканирование содержимого ZIP-архива и заполнение
87.    * им комбинированного списка.
88.    */
89.    public void scanZipFile()
90.    {
91.      new SwingWorker<Void, String>()
92.        {
93.          protected Void doInBackground() throws Exception
94.          {
95.            ZipInputStream zin = new ZipInputStream(new FileInput Stream(zipname));
96.            ZipEntry entry;
97.            while ((entry = zin.getNextEntry()) != null)
98.            {
99.              publish(entry.getName());
100.             zin.closeEntry();
101.           }
102.           zin.close();
103.           return null;
104.         }
105.
106.         protected void process(List<String> names)
107.         {
108.          for (String name : names)
109.            fileCombo.addItem(name);
110.
111.         }
112.     }.execute();
113.   }
```

```
114.
115. /**
116.  * Загрузка файла из ZIP-архива в текстовую область
117.  * @param name имя файла из архива
118.  */
119.  public void loadZipFile(final String name)
120.  {
121.    fileCombo.setEnabled(false);
122.    fileText.setText("");
123.    new SwingWorker<Void, Void>()
124.    {
125.      protected Void doInBackground() throws Exception
126.      {
127.        try
128.        {
129.         ZipInputStream zin = new ZipInputStream(new FileInputStream(zipname));
130.         ZipEntry entry;
131.
132.         // поиск записи с соответствующим именем в архиве
133.         while ((entry = zin.getNextEntry()) != null)
134.         {
135.          if (entry.getName().equals(name))
136.          {
137.            // считывание записи в текстовую область
138.            Scanner in = new Scanner(zin);
139.            while (in.hasNextLine())
140.            {
141.              fileText.append(in.nextLine());
142.              fileText.append("\n");
143.            }
144.          }
145.          zin.closeEntry();
146.         }
147.         zin.close();
148.        }
149.        catch (IOException e)
150.        {
151.          e.printStackTrace();
152.        }
153.        return null;
154.      }
155.
156.      protected void done()
157.      {
158.        fileCombo.setEnabled(true);
159.      }
160.    }.execute();
161.  }
162.
163.  public static final int DEFAULT_WIDTH = 400;
164.  public static final int DEFAULT_HEIGHT = 300;
165.  private JComboBox fileCombo;
166.  private JTextArea fileText;
167.  private String zipname;
168. }
```

API java.util.zip.ZipInputStream 1.1

- `ZipInputStream(InputStream in)`

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

- `ZipEntry getNextEntry()`

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

- `void closeEntry()`

  Закрывает текущую открытую запись в ZIP-файле. Далее может выполняться считывание следующей записи с использованием метода `getNextEntry()`.

API java.util.zip.ZipOutputStream 1.1

- `ZipOutputStream(OutputStream out)`

  Создает объект `ZipOutputStream`, позволяющий записывать сжатые данные в указанный объект `OutputStream`.

- `void putNextEntry(ZipEntry ze)`

  Записывает информацию из заданной записи `ZipEntry` в поток и позиционирует поток для данных. Далее данные могут записываться в этот поток с помощью метода `write()`.

- `void closeEntry()`

  Закрывает текущую открытую запись в ZIP-файле. Для начала работы со следующей записью нужно использовать метод `putNextEntry`.

- `void setLevel(int level)`

  Устанавливает метод сжатия, который должен применяться для данного потока `ZipOutputStream` в случае поступления записей без указанного метода.

  Параметры: `method` Метод сжатия, либо `DEFLATED`, либо `STORED`.

API java.util.zip.ZipEntry 1.1

- `ZipEntry(String name)`

  Параметры: `name` Имя записи.

- `long getCrc()`

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

- `String getName()`

  Возвращает имя данной записи.

- `long getSize()`

  Возвращает размер данной записи без сжатия или значение `-1`, если размер без сжатия не известен.

- `boolean isDirectory()`

  Возвращает `true`, если данная запись является каталогом.

- `void setMethod(int method)`

  Параметры: `method` Метод сжатия для записи; это обязательно должен быть либо `DEFLATED`, либо `STORED`.

- `void setSize(long size)`

  Устанавливает размер данной записи. Требуется только в том случае, если в роли метода сжатия выступает метод `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 в роли метода сжатия выступает метод `STORED`.

  Параметры: `crc` Контрольная сумма данной записи.

API java.util.zip.ZipFile 1.1

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

  Создает объект `ZipFile` для выполнения считывания из заданной строки или объекта `File`.

- `Enumeration entries()`

  Возвращает объект `Enumeration`, который перечисляет объекты `ZipEntry`, описывающие записи из `ZipFile`.

- `ZipEntry getEntry(String name)`

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

  Параметры: `name` Имя записи

- `InputStream getInputStream(ZipEntry ze)`

  Возвращает объект `InputStream` для заданной записи.

  Параметры: `ze` Запись `ZipEntry` в ZIP-файле

- `String getName()`

  Возвращает путь к данному ZIP-файлу.

Потоки объектов и сериализация

Использование формата записей фиксированной длины является хорошим вариантом, если сохранять требуется объекты одинакового типа. Однако объекты, создаваемые в объектно-ориентированных программах, редко бывают одного и того же типа. Например, может существовать массив под названием `staff`, номинально представляющий собой массив записей `Employee`, но при этом содержащий объекты, которые, по сути, являются экземплярами какого-нибудь подкласса, вроде `Manager`.

Конечно, можно было бы отыскать такой формат данных, который бы позволял сохранять подобные полиморфные коллекции, но, к счастью, в этом нет абсолютно никакой необходимости. Язык Java поддерживает универсальный механизм, который называется *сериализацией объектов* и предоставляет возможность записывать любой объект в поток и позже снова его считывать. (О том, откуда происходит термин "сериализация", более подробно будет объясняться чуть позже в этой главе.)

Для сохранения объектных данных, прежде всего, потребуется открыть объект `ObjectOutputStream`:

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

Далее, чтобы сохранить объект, остается просто воспользоваться методом `writeObject` класса `ObjectOutputStream` так, как показано в следующем 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);
```

Для выполнения обратного считывания объекта сначала нужно получить объект `ObjectInputStream`:

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

Затем остается просто извлечь объекты в том же порядке, в котором они записывались, с помощью метода `readObject`:

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

Имеется, однако, одно изменение, которое необходимо вносить в любой класс, который требуется сохранить и восстановить в объектном потоке, а именно — каждый такой класс должен обязательно реализовать интерфейс `Serializable`:

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

У интерфейса `Serializable` нет никаких методов, поэтому изменять свои собственные классы каким-либо образом разработчику не нужно. В этом отношении данный интерфейс похож на интерфейс `Cloneable`, о котором рассказывалось в главе 6 первого тома этой книги. Однако для того чтобы сделать класс пригодным для клонирования, все равно требовалось переопределять метод `clone` класса `Object`. А для того, чтобы сделать класс пригодным для сериализации, не требуется делать больше *ничего*.



Записывать и считывать только *объекты* можно с помощью методов `writeObject` и `readObject`. Что касается значений простых типов, то для них нужно применять такие методы, как `writeInt/readInt` или `writeDouble/readDouble`. (Классы объектных потоков реализуют интерфейсы `DataInput` и `DataOutput`.)

Класс `ObjectOutputStream` "за кулисами" просматривает все поля объектов и сохраняет их содержимое. Например, при записи объекта `Employee` в поток выходных данных будет записываться содержимое поля имени, поля даты и поля зарплаты.

Есть, однако, один важный вопрос, который нужно обязательно рассмотреть, и формулируется он следующим образом: "Что будет происходить в случае совместного использования одного и того же объекта в виде части состояния сразу несколькими другими объектами?".

Чтобы проиллюстрировать данную проблему, давайте внесем одно небольшое изменение в класс `Manager`. В частности, предположим, что у каждого менеджера есть секретарь:

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

Каждый объект `Manager` теперь будет содержать ссылку на объект `Employee`, описывающий секретаря. Конечно, два менеджера вполне могут пользоваться услугами одного и того же секретаря, как показано на рис. 1.6 и в следующем коде:

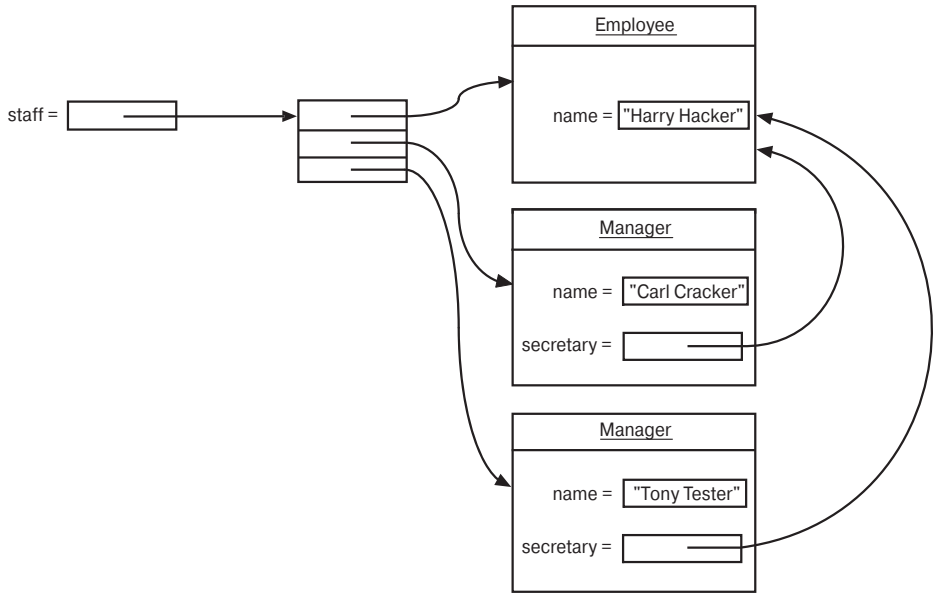


Рис. 1.6. Два менеджера могут совместно пользоваться услугами одного и того же сотрудника

```
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 цепочки объектов является непростой задачей. Конечно же, сохранять и восстанавливать адреса ячеек памяти для объектов секретарей мы не можем. При повторной загрузке каждый такой объект наверняка будет занимать уже совершенно другую ячейку памяти по сравнению с той, что он занимал первоначально.

Поэтому мы сохраняем каждый такой объект с *серийным номером* (serial number), откуда, собственно говоря, и происходит название механизма сериализации объектов. В общем, алгоритм выглядит следующим образом.

- Мы ассоциируем серийный номер с каждой встречающейся ссылкой на объект (как показано на рис. 1.7).
- Когда ссылка на объект встречается впервые, сохраняем данные этого объекта в поток.
- Если они уже были сохранены ранее, просто добавляем метку "same as previously saved object with serial number *x*" ("совпадает с уже сохранявшимся ранее объектом с серийным номером *x*").

При обратном считывании объектов процедура выполняется в обратном порядке.

- Когда объект указывается в потоке впервые, создаем его, инициализируем с данными потока и запоминаем ассоциацию между порядковым номером и объектной ссылкой.
- Когда встречается метка "same as previously saved object with serial number *x*", извлекаем ссылку на объект, соответствующий данному порядковому номеру.

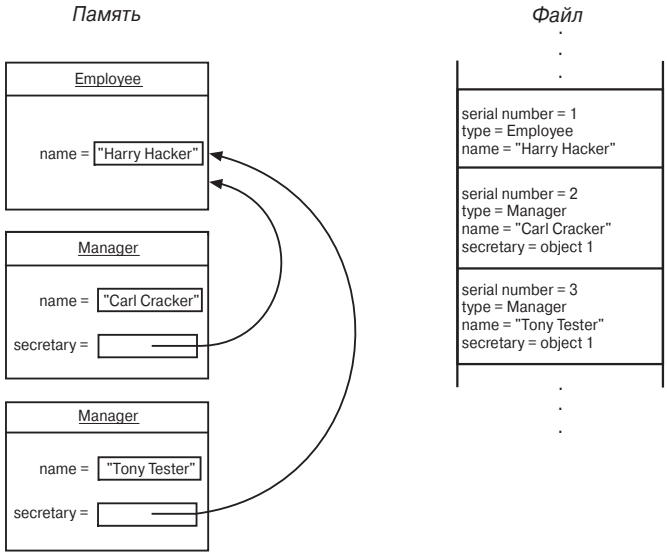


Рис. 1.7. Пример сериализации объектов



В этой главе демонстрируется то, как механизм сериализации можно применять для сохранения объектов в файле на диске и извлечения их в точности тем же образом, которым они и сохранялись. Другой очень важной областью применения этого механизма является передача коллекции объектов по сетевому соединению на другой компьютер. Необработанные адреса ячеек памяти являются такими же бессмысленными при взаимодействии с другим процессором, как и при работе с файлами. Благодаря замене адресов ячеек памяти серийными номерами механизм сериализации делает вполне возможной транспортировку коллекций объектов с одной машины на другую. Об этом способе применения механизма сериализации будет более подробно рассказываться в главе 5, при рассмотрении способов вызова удаленных методов.

В листинге 1.4 приведен код программы, способной сохранять и заново загружать цепочку объектов `Employee` и `Manager` (некоторые из которых имеют в качестве секретаря одного и того же сотрудника). Обратите внимание на то, что объект секретаря остается уникальным после повторной загрузки, т.е. при получении `newStaff[1]` повышения, которое отражается в полях `secretary` объектов `Manager`.

Листинг 1.4. Содержимое файла `ObjectStreamTest.java`

```
1.  import java.io.*;
2.  import java.util.*;
3.
4.  /**
5.   * @version 1.10 17 Aug 1998
6.   * @author Cay Horstmann
7.   */
8.  class ObjectStreamTest
9.  {
10.   public static void main(String[] args)
11.   {
12.     Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
13.     Manager carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
```

```
14.     carl.setSecretary(harry);
15.     Manager tony = new Manager("Tony Tester", 40000, 1990, 3, 15);
16.     tony.setSecretary(harry);
17.
18.     Employee[] staff = new Employee[3];
19.
20.     staff[0] = carl;
21.     staff[1] = harry;
22.     staff[2] = tony;
23.
24.     try
25.     {
26.       // сохранение всех записей о сотрудниках в файле employee.dat
27.       ObjectOutputStream out =
28.         new ObjectOutputStream(new FileOutputStream("employee.dat"));
29.       out.writeObject(staff);
30.       out.close();
31.
32.       // извлечение всех записей в новый массив
33.       ObjectInputStream in =
34.         new ObjectInputStream(new FileInputStream("employee.dat"));
35.       Employee[] newStaff = (Employee[]) in.readObject();
36.       in.close();
37.
38.        // повышение зарплаты секретаря
39.         newStaff[1].raiseSalary(10);
40.
41.        // распечатка только что считанных записей о сотрудниках
42.        for (Employee e : newStaff)
43.        System.out.println(e);
44.      }
45.      catch (Exception e)
46.      {
47.        e.printStackTrace();
48.      }
49.    }
50.  }
51.  class Employee implements Serializable
52.  {
53.    public Employee()
54.    {
55.    }
56.    public Employee(String n, double s, int year, int month, int day)
57.    {
58.      name = n;
59.      salary = s;
60.      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
61.      hireDay = calendar.getTime();
62.    }
63.    public String getName()
64.    {
65.      return name;
66.    }
67.    public double getSalary()
68.    {
69.      return salary;
70.    }
```

```
71.   public Date getHireDay()
72.   {
73.     return hireDay;
74.   }
75.
76.   public void raiseSalary(double byPercent)
77.   {
78.     double raise = salary * byPercent / 100;
79.     salary += raise;
80.   }
81.
82.   public String toString()
83.   {
84.     return getClass().getName() + "[name=" + name + ",salary=" + salary + ",hireDay="
85.      + hireDay + "]";
86.   }
87.
88.   private String name;
89.   private double salary;
90.   private Date hireDay;
91. }
92.
93. class Manager extends Employee
94. {
95. /**
96.  * Создание менеджера без секретаря
97.  * @param n имя сотрудника
98.  * @param s зарплата
99.  * @param year год приема на работу
100. * @param месяц приема на работу
101. * @param день приема на работу
102. */
103.
104. public Manager(String n, double s, int year, int month, int day)
105. {
106.   super(n, s, year, month, day);
107.    secretary = null;
108. }
109.
110. /**
111.  * Назначение менеджеру секретаря
112.  * @param s секретарь
113.  */
114. public void setSecretary(Employee s)
115. {
116.    secretary = s;
117.  }
118.
119. public String toString()
120. {
121.    return super.toString() + "[secretary=" + secretary + "]";
122. }
123.
124. private Employee secretary;
125. }
```

API java.io.ObjectOutputStream 1.1

- `ObjectOutputStream(OutputStream out)`

  Создание потока `ObjectOutputStream` для того, чтобы объекты можно было записывать в указываемый поток `OutputStream`.

- `void writeObject(Object obj)`

  Записывает указываемый объект в `ObjectOutputStream`. Этот метод сохраняет класс объекта, его сигнатуру и значения из любого нестатического и непереходного поля этого класса и его подклассов.

API java.io.ObjectInputStream 1.1

- `ObjectInputStream(InputStream in)`

  Создает поток `ObjectInputStream` для обратного считывания информации объектов из указанного потока `InputStream`.

- `Object readObject()`

  Считывает объект из потока `ObjectInputStream`. В частности, этот метод выполняет обратное считывание класса объекта, его сигнатуры, а также значений всех непереходных и нестатических полей этого класса и всех его подклассов. Он осуществляет десериализацию для предоставления возможности восстановления множества объектных ссылок.

Понимание формата файлов, создаваемых при сериализации объектов

При сериализации объектов их данные сохраняются в файле определенного формата. Конечно, можно использовать методы `writeObject/readObject`, которые не требуют знания того, как точно выглядит последовательность байтов, представляющая объекты в файле. Однако мы считаем, что изучение формата данных является чрезвычайно полезным для получения представления о процессе потоковой обработки объектов. Рассматриваемый здесь материал носит несколько технический характер, поэтому те, кого не интересуют детали реализации, могут 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 выглядит так:

```
00 05
```

(Для обозначения байтов везде в этом разделе будут использоваться шестнадцатеричные числа.) Далее в нем содержится последовательность объектов, которые следуют друг за другом именно в том порядке, в каком они сохранялись.

Строковые объекты сохраняются в следующем виде:

```
74    Обозначающее длину двухбайтовое число    Символы
```

Например, строка `Harry` будет иметь такой вид:

```
74 00 05 Harry
```

Unicode-символы строковых объектов сохраняются в “модифицированном формате UTF-8”.

При сохранении объекта также обязательно должен сохраняться и его класс. Описание класса включает:

- Имя класса.
- *Уникальный серийный идентификатор версии* (serial version unique ID), представляющий собой отпечаток (fingerprint) типов полей данных и сигнатур методов.
- Набор флагов, описывающий метод сериализации.
- Описание полей данных.

Отпечаток получается путем упорядочивания описаний класса, суперкласса, интерфейсов, типов полей и сигнатур методов каноническим образом и последующего применения к этим данным так называемого алгоритма SHA (Secure Hash Algorithm — алгоритм безопасного хеширования).

Алгоритм SHA позволяет быстро получать "отпечаток" большого блока информации. Этот отпечаток всегда представляет собой 20-байтовый пакет данных, каким бы ни был размер исходных данных. Он создается посредством выполнения над данными некой хитрой последовательности битовых операций, что дает практически 100%-ную уверенность в том, что в случае изменения информации хоть каким-либо образом, отпечаток тоже обязательно изменится. (Более подробную информацию о SHA можно найти в книге Уильяма Столингса (William Stallings) под названием *Cryptography and Network Security: Principles and Practice* (Prentice Hall, 2002 г.).) Механизм сериализации, однако, использует в качестве отпечатка класса только первых 8 байтов кода SHA. Это все равно позволяет быть уверенным в том, что в случае изменения полей данных или методов, отпечаток класса тоже изменится.

При считывании объекта его отпечаток сравнивается с текущим отпечатком класса. Если они не совпадают, это означает, что после записи объекта определение класса изменилось, и потому генерируется исключение. Конечно, на практике классы действительно часто совершенствуются, из-за чего может быть необходимо, чтобы программа умела считывать и более старые версии объектов. Более подробно этот вопрос будет рассматриваться чуть позже в этой главе, в разделе "Управление версиями".

В общем, идентификатор класса сохраняется в следующем виде:

```
72
2-байтовое число, обозначающее длину имени класса
Имя класса
8-байтовый отпечаток
1-байтовый флаг
2-байтовое число, представляющее количество дескрипторов полей данных
Дескрипторы полей данный
78 (маркер конца)
Тип суперкласса (если таковой отсутствует, тогда 70)
```

Байт флага состоит из трех битовых масок, определенных в `java.io.ObjectStreamConstants`:

```
static final byte SC_WRITE_METHOD = 1;
  // класс имеет метод writeObject, который записывает дополнительные данные
static final byte SC_SERIALIZABLE = 2;
  // класс реализует интерфейс Serializable
static final byte SC_EXTERNALIZABLE = 4;
  // класс реализует интерфейс Externalizable
```

Об интерфейсе `Externalizable` более подробно будет рассказываться чуть позже в этой главе. Пока что главное знать, что классы `Externalizable` предоставляют специальные методы для чтения и записи, которые используют выходные данные своих

собственных полей экземпляра. Рассматриваемые здесь классы реализуют интерфейс `Serializable` и имеют значение флага `02`. Класс `java.util.Date` тоже реализует интерфейс `Serializable`, но помимо этого еще также определяет свои собственные методы `readObject/writeObject` и имеет значение флага `03`.

Каждый дескриптор поля данных имеет следующий формат:

```
1-байтовый (т.е. занимающий 1 байт) код типа
2-байтовое (т.е. занимающее 2 байта) число, обозначающее длину имени поля
Имя поля
Имя класса (если поле является объектом)
```

Код типа может иметь одно из следующих обозначений:

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

Когда код типа выглядит как `L`, за именем поля следует тип поля. Строки имен классов и полей не начинаются строковым кодом `74`, а вот типы полей начинаются. Для названий (имен) типов полей используется несколько иная кодировка, а именно — формат платформо-ориентированных методов. Например, поле зарплаты класса `Employee` будет кодироваться следующим образом:

```
D 00 06 salary
```

Ниже показано, как будет полностью выглядеть дескриптор класса `Employee`:

`72 00 08 Employee`	
`E6 D2 86 7D AE AC 18 1B 02`	Отпечаток и флаги
`00 03`	Количество полей экзмепляра
`D 00 06 salary` (*зарплата*)	Тип и имя поля экземпляра
`L 00 07 hireDay` (*день приема на работу*)	Тип и имя поля экземпляра
`74 00 10 Ljava/util/Date;` (*Дата*)	Имя класса поля экземпляра — `Date`
`L 00 04 name`	Тип и имя поля экземпляра
`74 00 12 Ljava/lang/String;` (*строка*)	Имя класса поля экземпляра — `String`
`78`	Маркер конца
`70`	Суперкласс отсутствует

Эти дескрипторы являются довольно длинными. Если дескриптор *того же* класса требуется в файле еще раз, тогда уже используется сокращенная форма:

```
71     4-байтовый порядковый номер
```

Порядковый номер ссылается на приводившийся выше явный дескриптор класса. Схема нумерации будет рассматриваться чуть позже.

Объект сохраняется следующим образом:

```
73    Дескриптор класса    Данные объекта
```

Например, ниже показано, как будет сохраняться объект `Employee`:

`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 чуть позже
`74 00 0C Harry Hacker`	Значение поля `name` — `String`

Как видно, в файле данных сохраняется достаточно информации для восстановления объекта `Employee`.

Массивы сохраняются в следующем формате:

```
75 Дескриптор класса  4-байтовое число, указывающее на общее количество записей  Записи
```

Имя класса массива в дескрипторе класса имеет тот же формат, что и в платформо-ориентированных методах (который немного отличается от формата, используемого для имен классов в других дескрипторах классов). В этом формате имена классов начинаются с `L`, а оканчиваются символом точки с запятой.

Например, массив из трех объектов `Employee` будет начинаться так:

```
75                                  Массив
  72 00 0B [LEmployee;              Новый класс, длина строки, имя класса Employee[]
     FC BF 36 11 C5 91 11 C7 02     Отпечаток и флаги
     00 00                          Количество полей экземпляра
     78                             Маркер конца
     70                             Суперкласс отсутствует
     00 00 00 03                    Количество записей в массиве
```

Обратите внимание на то, что отпечаток массива объектов `Employee` отличается от отпечатка самого класса `Employee`.

Всем объектам (массивам и строкам включительно) и всем дескрипторам классов при сохранении в файле выходных данных присваиваются порядковые номера. Начинаются все эти номера с комбинации `00 7E 00 00`.

Ранее уже упоминалось о том, что весь целиком дескриптор любого конкретного класса встречается только один раз. Все последующие дескрипторы просто ссылаются на него. Например, в предыдущем примере повторяющаяся ссылка на класс `Date` кодировалась следующим образом:

```
71 00 7E 00 08
```

Тот же самый механизм используется и для объектов. Если записывается ссылка на сохраненный ранее объект, она сохраняется точно таким же образом, т.е. в виде маркера `71`, со следующим за ним порядковым номером. То, указывает данная ссылка на объект или на дескриптор класса, всегда можно понять по контексту.

И, наконец, нулевая ссылка сохраняется следующим образом:

```
70
```

Ниже приведен вывод демонстрировавшейся в предыдущем разделе программы `ObjectRefTest` с комментариями. Если хотите, можете запустить эту программу, посмотреть на шестнадцатеричный дамп ее файла данных `employee.dat` и сравнить его с этим приведенным выводом. 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 флаги
    00 00                           Количество полей экземпляра
    78                              Маркер конца
    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 флаги
       00 01                        Количество полей данных
```

```
  L 00 09 secretary                 Тип и имя поля экземпляра
  74 00 0A LEmployee;               Имя класса поля экземпляра — String (порядковый номер #3)
  78                                Маркер конца
  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 флаги
    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 поля экземпляра — String (порядковый номер #5)
    L 00 04 name                    Тип и имя поля экземпляра
    74 00 12 Ljava/lang/String;     Имя класса поля экземпляра — String (порядковый номер #6)
    78                              Маркер конца
    70                              Суперкласс отсутствует
  40 F3 88 00 00 00 00 00           Значение поля salary — double
  73                                Значение поля hireDay — 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 флаги
      00 00                         Переменные экземпляра отсутствуют
      78                            Маркер конца
      70                            Суперкласс отсутствует
    77 08                           Внешнее хранилище, количество байтов
    00 00 00 83 E9 39 E0 00         Дата
    78                              Маркер конца
  74 00 0C Carl Cracker             Значение поля — 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 #12)
      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                              Маркер конца
    74 00 0C Harry Hacker           Значение поля name — String (порядковый номер #13)
71 00 7E 00 0B                      staff[1] — существующий объект (нужно
                                        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                              Маркер конца
  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 может столь же увлекательно, как чтение телефонного справочника. Знать точный формат файла не обязательно (если только не предпринимается попытка добиться какого-то зловредного эффекта путем изменения данных), но понимать, что объектный поток содержит детальное описание всех входящих в него объектов вместе с вполне достаточным количеством информации для проведения восстановления как объектов, так и массивов объектов, конечно, все равно очень полезно.

В целом, главное запомнить следующее:

- объектный поток содержит информацию о типах и полях данных всех входящих в него объектов;
- каждому объекту назначается серийный (порядковый) номер;
- повторные вхождения того же самого объекта сохраняются в виде ссылок на его серийный номер.

Изменение механизма сериализации, используемого по умолчанию

Некоторые поля данных не должны никогда подвергаться сериализации, как, например, поля с целочисленными значениями, в которых хранятся дескрипторы файлов или дескрипторы окон, имеющие значение только для платформо-ориентированных методов. Такая информация гарантировано будет становиться бесполезной при повторной загрузке объекта в более позднее время или при его транспортировке на другую машину. На самом деле, некорректные значения для таких полей могут даже приводить к аварийному завершению работы платформо-ориентированных методов. В Java предусмотрен простой механизм, позволяющий предотвращать сериализацию подобных полей. Все, что требуется — это просто помечать их ключевым словом `transient` (переходное). Помечать поля этим ключевым словом также необходимо и в том случае, если они относятся к не сериализируемым классам. В случае сериализации объектов переходные поля всегда пропускаются.

Механизм сериализации предусматривает для отдельных классов возможность добавления к стандартному поведению чтения и записи процедуры проверки правильности данных или любого другого желаемого действия. Сериализируемый класс может определять методы со следующими сигнатурами:

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

Тогда поля данных больше не будут автоматически подвергаться сериализации, вместо этого будут уже вызываться данные методы.

Рассмотрим типичный пример. В пакете `java.awt.geom` некоторые классы, вроде `Point2D.Double`, не являются сериализируемыми. Теперь давайте предположим, что требуется сериализировать класс `LabeledPoint`, содержащий поля `String` и `Point2D.Double`. Чтобы сделать это, мы первым делом помечаем поле `Point2D.Double` как `transient` во избежание появления исключения `NotSerializableException`.

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

Далее в рамках метода `writeObject` мы сначала записываем дескриптор объекта и поле `String` (состояние) путем вызова метода `defaultWriteObject`. Это — специальный метод класса `ObjectOutputStream`, который может вызываться только из метода `writeObject` сериализируемого класса. Затем мы записываем координаты точки с помощью стандартных вызовов `DataOutput`.

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

В методе `readObject` осуществляется 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`, который предоставляет свои собственные методы `readObject` и `writeObject`. Эти методы записывают дату в виде количества миллисекунд от начальной точки отсчета (каковой является 1 января 1970 г., полночь по Гринвичу). Класс `Date` имеет сложное внутреннее представление, в котором хранится как объект `Calendar`, так и счетчик миллисекунд для оптимизации операций поиска. Состояние объекта `Calendar` является избыточным и потому не подлежит обязательному сохранению.

Методам `readObject` и `writeObject` необходимо сохранять и загружать поля только своих данных. Заботится о данных суперкласса или информации каких-либо других классов им не нужно.

Вместо того чтобы позволять механизму сериализации сохранять и восстанавливать объектные данные, класс может определять свой собственный механизм. Для этого он должен реализовать интерфейс `Externalizable`. Это, в свою очередь, требует, чтобы он еще также определял и 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 описывались 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. При считывании объекта типа `Externalizable` объектный поток создает этот объект с помощью используемого по умолчанию конструктора и затем вызывает метод `readExternal`. Ниже показано, как эти методы можно реализовать для класса `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());
}
```



Сериализация протекает несколько медленно, поскольку виртуальной машине требуется отыскивать структуру каждого объекта. Тем, кого волнует производительность и кому необходимо считывать и записывать большое количество объектов определенного класса, следует обязательно ознакомиться со способами применения интерфейса `Externalizable`. Технический анализ, результаты которого можно просмотреть по адресу `http://java.sun.com/developer/TechTips/2000/tt0425.html`, показал, что в случае класса `Employee`, при использовании внешних методов чтения и записи показатели производительности были примерно на 35–40% больше, чем при использовании стандартного механизма сериализации.



В отличие от методов `readObject` и `writeObject`, которые являются приватными и могут вызываться только механизмом сериализации, методы `readExternal` и `writeExternal` являются общедоступными. В частности, метод `readExternal` потенциально допускает изменение состояния существующего объекта.

Сериализация одноэлементных множеств и перечислений, обеспечивающих безопасность типов

Особое внимание нужно проявлять при сериализации и десериализации объектов, которые считаются уникальными. Обычно подобное происходит при реализации одноэлементных множеств (singletons) и перечислений, обеспечивающих безопасность типов.

В случае использования конструкции `enum`, которая предлагается, начиная с версии Java SE 5.0, волноваться о сериализации не нужно: все будет работать так, как надо. Однако, что делать, если имеется некий унаследованный код, содержащий перечислимый тип вроде показанного ниже:

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

Применение подобного подхода было очень широко распространено до того, как в язык Java были добавлены перечисления. Обратите внимание, что конструктор является приватным. Следовательно, за пределами `Orientation.HORIZONTAL` и `Orientation.VERTICAL` больше не могут создаваться никакие объекты. В частности, для выполнения проверки на предмет равенства объектов можно использовать операцию ==:

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

Существует одна важная особенность, о которой следует обязательно помнить, когда обеспечивающее безопасность типов перечисление реализует интерфейс `Serializable`. Стандартный (предлагаемый по умолчанию) механизм сериализации в таком случае не подходит. Например, давайте попробуем записать значение типа `Orientation` и затем снова его прочитать:

```
Orientation original = Orientation.HORIZONTAL;
ObjectOutputStream out = . . .;
```

```
out.write(value);
out.close();
ObjectInputStream in = . . .;
Orientation saved = (Orientation) in.read();
```

Теперь проверка

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

завершится неудачей, потому что, фактически, значение `saved` является совершенно новым объектом типа `Orientation` и не равняется ни одной из предопределенных констант. Даже несмотря на то, что конструктор является приватным, механизм сериализации все равно может создавать новые объекты!

Для решения описанной проблемы нужно определять еще один специальный метод сериализации под названием `readResolve`. В случае определения метода `readResolve`, он вызывается после десериализации объекта. Он должен возвращать объект, затем превращаемый в возвращаемое значение метода `readObject`. В нашем случае метод `readResolve` будет инспектировать поле `value` и возвращать соответствующую перечислимую константу:

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

Важно не забывать добавлять метод `readResolve` во все перечисления, обеспечивающие безопасность типов, в унаследованном коде и во все классы, в которых применяется схема одноэлементных множеств.

Управление версиями

В случае применения сериализации для сохранения объектов разработчику придется задумываться о том, что будет происходить при усовершенствовании его программы. Сможет ли версия 1.1 читать старые файлы? Смогут ли пользователи, попрежнему работающие с версией 1.0, читать файлы, которые теперь генерирует новая версия? Конечно, было бы совсем неплохо, если бы файлы объектов могли успешно справляться с эволюцией классов.

На первый взгляд подобное кажется невозможным. Когда определение класса изменяется хоть каким-то образом, сразу же изменяется и его отпечаток SHA, а, как нам уже известно, объектные потоки будут отказываться считывать объекты с отличающимися отпечатками. Однако класс может сообщать о том, что он является *совместимым* с более ранней версией самого себя. Чтобы добиться такого эффекта, первым делом нужно получить отпечаток *более ранней* версии класса. Это можно сделать с помощью входящей в состав JDK автономной программы `serialver`. Например, выполнение следующей команды:

```
serialver Employee
```

приведет к получению такого вывода:

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

В случае запуска программы `serialver` с опцией `-show` она отобразит графическое диалоговое окно, показанное на рис. 1.8.

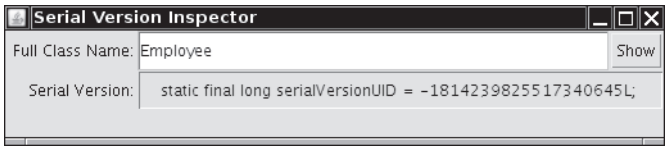


Рис. 1.8. Графическая версия программы `serialver`

Все *более поздние* версии класса должны определять константу `serialVersionUID` с точно таким же отпечатком, что и у оригинала.

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

При наличии у класса статической переменной-члена по имени `serialVersionUID`, он не будет вычислять отпечаток вручную, а будет просто использовать содержащееся в этом члене значение.

После размещения такого статического члена внутри класса, система сериализации будет сразу же готова к считыванию разных версий объектов этого класса.

В случае изменения только методов класса, проблем со считыванием данных новых объектов не будет. С другой стороны, в случае изменения полей данных кое-какие проблемы все-таки могут возникнуть. Например, старый объект файла может содержать больше или меньше полей данных, нежели тот, что сейчас применяется в программе, или же типы полей данных могут отличаться. В таком случае объектный поток будет пытаться преобразовать потоковый объект в текущую версию класса.

Объектный поток будет сравнивать поля данных текущей версии класса с полями данных той версии класса, которая указана в потоке. Конечно, принимать во внимание он будет только не переходные и не статические поля данных. В случае совпадения имен, но несовпадения типов полей, объектный поток, естественно, не будет пытаться преобразовать один тип в другой: данные объекты будут считаться просто не совместимыми. При наличии у объекта в потоке таких полей данных, которых нет в текущей версии, объектный поток 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 (каковым в случае объектов является `null`, в случае чисел — `0`, а в случае булевских значений — `false`).

Давайте рассмотрим пример. Предположим, что мы сохранили ряд записей о сотрудниках на диске с использованием исходной версии класса (версии 1.0), а затем заменили класс `Employee` версией 2.0, добавив в нее поле данных под названием `department`. На рис. 1.9 показано, что произойдет при считывании объекта версии 1.0 в программе, в которой используются объекты версии 2.0. А произойдет следующее: для поля `department` будет установлено значение `null`. На рис. 1.10 показан противоположный сценарий, т.е., что произойдет в случае считывания объекта версии 2.0 программой, в которой используются объекты версии 1.0. В этом случае дополнительное поле `department` будет просто проигнорировано.

Будет ли данный процесс безопасным? Все зависит от обстоятельств. Игнорирование поля данных кажется безвредным, ведь у получателя все равно остаются все те данные, с которым он знал, как работать. А вот установка для поля данных значения `null` может и не быть безопасным процессом.

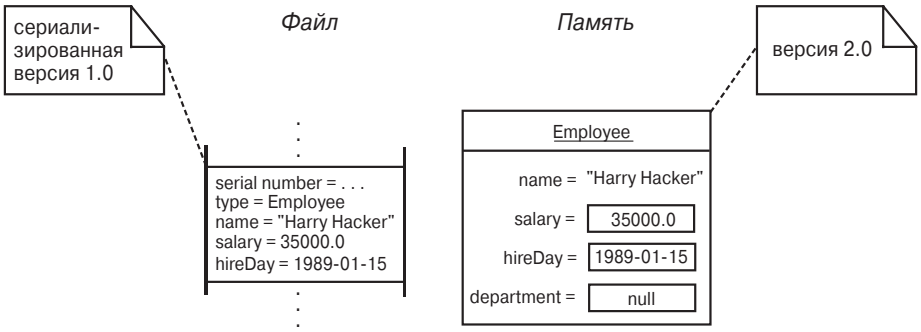


Рис. 1.9. Считывание объекта с меньшим количеством полей данных

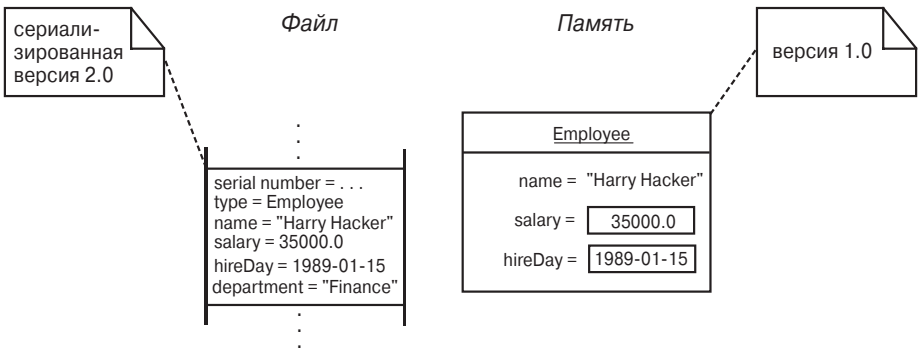


Рис. 1.10. Считывание объекта с большим количеством полей данных

Многие классы прилагают значительные усилия для инициализации всех полей данных во всех конструкторах значениями, отличными от `null`, чтобы методам не нужно было быть готовыми к обработке `null`-данных. В принципе разработчик классов должен сам решать, что ему лучше: реализовать дополнительный код в методе `readObject` для устранения проблем с совместимостью между версиями или удостоверяться в том, что методы являются достаточно надежными для того, что успешно обрабатывать `null`-данные.

Использование сериализации для клонирования

У механизма сериализации имеется еще один интересный способ применения: он позволяет легко клонировать объект при условии, что его класс является сериализируемым. Достаточно всего лишь сериализировать его в поток выходных данных, а затем прочитать обратно. В результате получится новый объект, представляющий собой точную (“глубокую”) копию того, что уже существует. Записывать этот объект в файл вовсе не обязательно: можно использовать класс `ByteArrayOutputStream` и сохранять данные в байтовый массив.

Как показывает код, приведенный в листинге 1.5, чтобы получить клон объекта (`clone`), достаточно просто расширить класс `SerialCloneable`.

Следует обязательно знать о том, что такой метод, хоть и является хитрым, обычно будет работать гораздо медленнее, чем метод клонирования, который явным образом создает новый объект и копирует или клонирует поля данных.

Листинг 1.5. Содержимое файла `SerialCloneTest.java`

```
1.  import java.io.*;
2.  import java.util.*;
3.
4.  public class SerialCloneTest
5.  {
6.    public static void main(String[] args)
7.    {
8.      Employee harry = new Employee("Harry Hacker", 35000, 1989, 10, 1);
9.      // клонирование harry
10.     Employee harry2 = (Employee) harry.clone();
11.
12.     // модификация harry
13.     harry.raiseSalary(10);
14.
15.     // теперь harry и его клон отличаются
16.     System.out.println(harry);
17.     System.out.println(harry2);
18.   }
19. }
20.
21. /**
22.   Класс, в методе клонирования которого применяется сериализация.
23. */
24.  class SerialCloneable implements Cloneable, Serializable
25.  {
26.    public Object clone()
27.    {
28.      try
29.      {
30.        // сохранение объекта в массиве байтов
31.        ByteArrayOutputStream bout = new ByteArrayOutputStream();
32.        ObjectOutputStream out = new ObjectOutputStream(bout);
33.        out.writeObject(this);
34.        out.close();
35.
36.        // считывание клона объекта из массива байтов
37.        ByteArrayInputStream bin = new ByteArrayInput Stream(bout.toByteArray());
38.        ObjectInputStream in = new ObjectInputStream(bin);
39.        Object ret = in.readObject();
40.        in.close();
41.
42.      return ret;
43.     }
44.     catch (Exception e)
45.    {
46.      return null;
47.    }
48.   }
49. }
50.
51. /**
52.   Уже знакомый класс Employee, переопределенный таким образом,
53.   что теперь он расширяет класс SerialCloneable.
54. */
```

```
55.  class Employee extends SerialCloneable
56.  {
57.    public Employee(String n, double s, int year, int month, int day)
58.    {
59.      name = n;
60.      salary = s;
61.      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
62.      hireDay = calendar.getTime();
63.    }
64.
65.    public String getName()
66.    {
67.     return name;
68.    }
69.
70.    public double getSalary()
71.    {
72.      return salary;
73.    }
74.
75.    public Date getHireDay()
76.    {
77.     return hireDay;
78.   }
79.
80.   public void raiseSalary(double byPercent)
81.   {
82.     double raise = salary * byPercent / 100;
83.     salary += raise;
84.   }
85.
86.   public String toString()
87.   {
88.      return getClass().getName()
89.        + "[name=" + name
90.        + ",salary=" + salary
91.        + ",hireDay=" + hireDay
92.        + "]";
93.   }
94.
95.   private String name;
96.   private double salary;
97.   private Date hireDay;
98.  }
```

Управление файлами

О том, как можно считывать и записывать данные из и в файл, мы уже говорили. Однако управление файлами не ограничивается только чтением и записью. Класс `File` инкапсулирует все функциональные возможности, которые могут потребоваться для работы с файловой системой на машине пользователя. Например, его можно применять для выяснения времени последнего изменения файла, а также удаления или переименования файла. Другими словами, классы потоков имеют дело с содержимым файла, а класс `File` — с хранением файла на диске.



Кроме того, как часто бывает в Java, класс `File` следует принципу “наименьшего знаменателя”. Например, в Windows можно узнавать о наличии у файла флага “только для чтения” (и даже устанавливать его), а вот хотя узнавать о том, является ли файл скрытым, можно, скрывать его без использования платформо-ориентированного метода нельзя.

Самый простейший конструктор объекта `File` принимает в качестве параметра (полное) имя файла. Если путь к этому файлу не указывается, Java использует текущий каталог. Например, строка кода

```
File f = new File("test.txt");
```

приведет к созданию объекта файла с именем `test.txt` в текущем каталоге. (Под “текущим каталогом” в данном случае подразумевается текущий каталог того процесса, который выполняет виртуальная машина. Если виртуальная машина была запущена из командной строки, это будет тот каталог, из которого был запущен исполняемый файл `java`.)



Поскольку символ обратной косой черты служит в строках Java символом отмены, при указании путей в стиле Windows следует обязательно не забывать использовать два символа обратной косой черты (`"C:\\Windows\\win.ini"`). В Windows можно еще также применять и один символ левой косой черты (`"C:/Windows/win.ini"`), поскольку в большинстве системных вызов, ориентированных на обработку файлов, символы левой косой черты будут интерпретироваться как разделители файлов. Однако делать это все-таки не рекомендуется, поскольку поведение системных функций Windows подвержено изменениям, а также потому, что в других операционных системах разделитель файлов может выглядеть и по-другому. Вместо этого при создании переносимых программ следует использовать символ разделителя файлов той платформы, на которой данная программа выполняется. Увидеть, как он выглядит, можно в строковой константе `File.separator`.

Вызов этого конструктора *не приводит к созданию файла с этим именем, если он не существует.* На самом деле создание файла из объекта `File` осуществляется с помощью либо одного из конструкторов классов потоков, либо метода `createNewFile` из класса `File`. Метод `createNewFile` создает файл только в том случае, если не существует ни одного файла с таким именем, и возвращает значение `boolean`, информирующее о том, удалось ли ему успешно завершить свою операцию.

С другой стороны, когда имеется объект `File`, о том, существует файл с таким именем или нет, в классе `File` сообщает метод `exists`. Например, следующая демонстрационная программа почти наверняка будет выводить на любом компьютере `false`, но при этом все равно сможет распечатывать путь к данному не существующему файлу.

```
import java.io.*;
public class Test
{
  public static void main(String args[])
  {
    File f = new File("afilethatprobablydoesntexist");
    System.out.println(f.getAbsolutePath());
    System.out.println(f.exists());
  }
}
```

Существует еще два других конструктора для объектов `File`.

Первый выглядит так:

```
File(String path, String name)
```

и создает объект File с заданным именем в том каталоге, на который указывает параметр path. (Если в параметре path содержится значение null, тогда он создает объект File в текущем каталоге.)

И, наконец, второй конструктор позволяет использовать существующий объект File:

```
File(File dir, String name)
```

Здесь объект File представляет каталог. Как и раньше, если в dir содержится значение null, конструктор создает объект File в текущем каталоге.

Немного путающим является то, что объект File может представлять как файл, так и каталог (пожалуй, из-за того, что в операционной системе, с которой более всего были знакомы разработчики Java, каталоги реализуются в виде файлов). Для указания того, файл или все-таки каталог представляет объект File, применяются методы isDirectory и isFile. Это удивляет, поскольку в объектно-ориентированной системе следовало бы ожидать наличие отдельного класса Directory, возможно, расширяющего класс File.

Сделать так, чтобы объект представлял каталог, можно просто путем указания в конструкторе File имени требуемого каталога:

```
File tempDir = new File(File.separator + "temp");
```

Если нужного каталога еще не существует, его можно создать с помощью метода mkdir:

```
tempDir.mkdir();
```

Если объект файла представляет каталог, то для получения массива имен содержащихся в этом каталоге файлов можно воспользоваться методом list(). В программе, код которой приведен в листинге 1.6, все эти методы применяются для отображения подструктуры каталогов любого вводимого в командной строке пути. (Эту программу можно довольно легко превратить в служебный класс, возвращающий список подкаталогов для дальнейшей обработки.)



Для манипулирования именами файлов и каталогов лучше всегда использовать объекты File, а не строки. Например, методу equals класса File известно, что в некоторых файловых системах регистру не придается значения, и что замыкающий символ / в имени каталога роли не играет.

Листинг 1.6. Содержимое файла FindDirectories.java

```
1.  import java.io.*;
2.
3.  /**
4.   * @version 1.00 05 Sep 1997
5.   * @author Gary Cornell
6.   */
7.  public class FindDirectories
8.  {
9.    public static void main(String[] args)
10.   {
11.     // в случае отсутствия аргументов начинаем с родительского каталога
12.     if (args.length == 0) args = new String[] { ".." };
```

```
13.
14.     try
15.     {
16.       File pathName = new File(args[0]);
17.       String[] fileNames = pathName.list();
18.
19.     // перечисляем все файлы в каталоге
20.     for (int i = 0; i < fileNames.length; i++)
21.     {
22.       File f = new File(pathName.getPath(), fileNames[i]);
23.
24.     // если файл опять-таки оказывается каталогом,
25.     // вызываем главный метод рекурсивным образом
26.     if (f.isDirectory())
27.     {
28.       System.out.println(f.getCanonicalPath());
29.       main(new String[] { f.getPath() });
30.     }
31.    }
32.   }
33.   catch (IOException e)
34.   {
35.     e.printStackTrace();
36.    }
37.   }
38.  }
```

Вместо того чтобы получать полный список всех файлов в каталоге, можно сужать этот список необходимым образом путем передачи методу `list` в качестве параметра объекта `FileNameFilter`. Такие объекты представляют собой просто экземпляры класса, удовлетворяющего требованиям интерфейса `FilenameFilter`.

Для реализации интерфейса `FilenameFilter` классу требуется всего лишь определить метод под названием `accept`. Ниже приведен пример простого класса `FilenameFilter`, разрешающего принимать только файлы с указанным расширением:

```
public class ExtensionFilter implements FilenameFilter
{
  public ExtensionFilter(String ext)
  {
    extension = "." + ext;
  }
  public boolean accept(File dir, String name)
  {
    return name.endsWith(extension);
  }
  private String extension;
}
```

При написании переносимых программ непростой задачей является указание имен файлов с подкаталогами. Как уже упоминалось ранее, оказывается, что символ левой косой черты (стандартный разделитель в UNIX) в Windows тоже может применяться в качестве разделителя каталогов, однако в других операционных системах подобное может быть недопустимо, поэтому пользоваться этим символом мы не рекомендуем.



Тем, кто все-таки хочет применять символ левой косой черты в качестве разделителя каталогов в Windows при создании объектов `File`, следует знать, что метод `getAbsolutePath` будет возвращать имя файла, содержащее символы левой косой черты, что, естественно, будет выглядеть странным для пользователей Windows. Поэтому вместо него следует использовать метод `getCanonicalPath`, который будет заменять все символы левой косой черты символами обратной косой черты.

Гораздо лучше использовать информацию о текущем разделите каталогов, которую класс `File` хранит в статическом поле экземпляра под названием `separator`. В Windows это будет символ обратной косой черты (\), а в UNIX — символ левой косой черты (/). Например:

```
File foo = new File("Documents" + File.separator + "data.txt")
```

Конечно, если воспользоваться второй альтернативной версией конструктора `File`:

```
File foo = new File("Documents", "data.txt")
```

тогда сам конструктор будет предоставлять правильный разделитель.

В приведенных ниже врезках API представлены остальные, наиболее важные с нашей точки зрения методы класса `File`, применение которых не должно вызывать никаких сложностей.

API java.io.File 1.0

- `boolean canRead()`
- `boolean canWrite()`
- `boolean canExecute()` 6

  Указывает, является ли файл доступным для чтения, записи или выполнения.

- `boolean setReadable(boolean state, boolean ownerOnly)` 6
- `boolean setWritable(boolean state, boolean ownerOnly)` 6
- `boolean setExecutable(boolean state, boolean ownerOnly)` 6

  Устанавливает для данного файла состояние доступности для чтения, записи или выполнения. Если `ownerOnly` равно `true`, выбранное состояние устанавливается только для владельца файла. В противном случае оно устанавливается для всех. Если установка состояния проходит успешно, данные методы возвращают `true`.

- `static boolean createTempFile(String prefix, String suffix)` 1.2
- `static boolean createTempFile(String prefix, String suffix, File directory)` 1.2

  Создает в стандартном каталоге временных файлов системы или в указанном каталоге временный файл, используя для генерации временного имени заданный префикс и суффикс.

  Параметры: `prefix` Строка префикса, состоящая как минимум из трех символов.

  `suffix` Необязательный суффикс. В случае указания `null` используется суффикс `.tmp`.

  `directory` Каталог, в котором создается файл. В случае указания `null` файл создается в текущем рабочем каталоге.

- `boolean delete()`

  Пытается удалить файл. Если ему это удается, возвращает `true`, в противном случае возвращает `false`.

- `void deleteOnExit()`

  Требует, чтобы файл был удален при завершении работы виртуальной машины.

- `boolean exists()`

  Если каталог существует, возвращает `true`, в противном случае — `false`.

- `String getAbsolutePath()`

  Возвращает строку, в которой содержится абсолютный путь. Совет: лучше использовать вместо этого метода метод `getCanonicalPath`.

- `File getCanonicalFile()` 1.2

  Возвращает объект `File`, содержащий канонический путь к файлу. В частности, он удаляет из пути все избыточные каталоги "`.`", использует правильный разделитель для имен каталогов и применяет такую схему выделения заглавными буквами, которая наиболее всего подходит для лежащей в основе файловой системы.

- `String getCanonicalPath()` 1.1

  Возвращает строку, содержащую канонический путь. В частности, он удаляет из пути все избыточные каталоги "`.`", использует правильный разделитель для имен каталогов и применяет такую схему выделения заглавными буквами, которая наиболее всего подходит для лежащей в основе файловой системы.

- `String getName()`

  Возвращает строку, содержащую имя файла для данного объекта `File` (без остальной части пути).

- `String getParent()`

  Возвращает строку, содержащую имя родителя данного объекта `File`. Если за объектом `File` скрывается файл, тогда этим родителем является содержащий данный файл каталог, а если за ним скрывается каталог, тогда — либо имя родительского каталога, либо значение `null`, если родительского каталога не существует.

- `File getParentFile()` 1.2

  Возвращает объект `File` для родителя данного каталога `File`. Определение "родителя" приводилось в описании метода `getParent`.

- `String getPath()`

  Возвращает строку, содержащую путь к файлу.

- `boolean isDirectory()`

  Возвращает `true`, если объект `File` представляет собой каталог, в противном случае возвращает `false`.

- `boolean isFile()`

  Возвращает `true`, если объект `File` представляет собой файл, а не, например, каталог или устройство.

- `boolean isHidden()` 1.2

  Возвращает `true`, если объект `File` представляет собой скрытый файл или каталог.

- `long lastModified()`

  Возвращает время последнего изменения файла (отсчитываемое в миллисекундах, начиная с полуночи 1 января 1970 г. по Гринвичу) или 0, если такого файла не существует. Для преобразования этого значения в дату можно использовать метод `Date(long)`.

- `long length()`

  Возвращает длину файла в байтах или `0`, если такого файла не существует.

- `String[] list()`

  Возвращает массив строк, в которых содержатся имена файлов и каталогов, представленных в данном объекте `File`, или значение `null`, если данный объект `File` не является каталогом.

- `String[] list(FilenameFilter filter)`

  Возвращает массив имен файлов и каталогов, содержащихся в данном объекте `File`, которые удовлетворяют условиями фильтра, или значение `null`, если таковых в нем не существует.

- `File[] listFiles()` 1.2

  Возвращает массив объектов `File`, соответствующих файлам и каталогам, которые содержатся в данном объекте `File`, или значение `null`, если данный объект `File` не является каталогом.

- `File[] listFiles(FilenameFilter filter)` 1.2

  Возвращает массив объектов `File` для тех содержащихся в данном объекте `File` файлов и каталогов, которые удовлетворяют условиям фильтра, или значение `null`, если в нем таковых не существует.

- `static File[] listRoots()` 1.2

  Возвращает массив объектов `File`, соответствующих всем доступным корневым узлам файлов. (Например, в системе Windows это будут объекты `File`, представляющие установленные диски, причем как локальные, так и отображенные сетевые, а в системе UNIX это будут просто символы `/`.)

- `boolean createNewFile()` 1.2

  Автоматически создает новый файл с предоставляемым объектом `File` именем, если файла с таким именем не существует. Выполнение операций проверки на предмет наличия файла с таким именем и непосредственно создания этого файла не может прерываться никакими другими процессами файловой системы. Если файл был создан, возвращает `true`.

- `boolean mkdir()`

  Создает подкаталог с предоставляемым объектом `File` именем. Если каталог был успешно создан, возвращает `true`, в противном случае возвращает `false`.

- `boolean mkdirs()`

  В отличие от `mkdir`, создает родительские каталоги, если в них есть необходимость. Если ему не удается создать хотя бы один из необходимых каталогов, возвращает `false`.

- `boolean renameTo(File newName)`

  Возвращает `true`, если имя было изменено, в противном случае возвращает `false`.

- `boolean setLastModified(long time)` 1.2

  Устанавливает время последнего изменения файла. Если это удается, возвращает `true`, в противном случае возвращает `false`. Параметр `time` — это длинное целое, представляющее количество миллисекунд, которое прошло, начиная с полуночи 1 января 1970 г. по Гринвичу. Для вычисления этого значения применяется метод `getTime` класса `Date`.

- `boolean setReadOnly()` 1.2

  Делает файл доступным только для чтения. Если это удается, возвращает `true`, в противном случае возвращает `false`.

- `URL toURL()` 1.2

  Преобразует объект `File` в объект `URL`.

- `long getTotalSpace()` 6
- `long getFreeSpace()` 6
- `long getUsableSpace()` 6

  Получают общий размер, количество невыделенных байтов и количество доступных байтов на диске, описываемом данным объектом `File`. Если данный объект `File` описывает не раздел, возвращают `0`.

API `java.io.FilenameFilter 1.0`

- `boolean accept(File dir, String name)`

  Должен определяться так, чтобы в случае, если файл отвечает критериям фильтра, возвращалось значение `true`.

Параметры:	`dir`	Объект `File`, представляющий каталог, в котором содержится нужный файл.
	`name`	Имя файла.

Новые средства для обработки ввода-вывода

В версии Java SE 1.4 был представлен ряд улучшенных возможностей для обработки ввода-вывода, которые в общем были названы "новыми средствами для обработки ввода-вывода" (new I/O) и объединены в пакет `java.nio`. (Конечно, прозвище "новые" уже немного устарело, потому что прошло несколько лет, и данный пакет новым уже больше не является.)

В частности, в состав этого пакета входит поддержка для следующих средств:

- кодирования и декодирования наборов символов;
- неблокирующего ввода-вывода;
- отображаемых в памяти файлов;
- блокировки файлов.

О кодировании и декодировании символов уже рассказывалось ранее в этой главе, в разделе "Наборы символов". О неблокирующих средствах ввода-вывода речь пойдет в главе 3, потому что они играют особо важную роль при передаче данных по сети. Следовательно, остается рассмотреть только отображаемые в памяти файлы и блокировку файлов, что и будет сделано в следующих разделах.

Отображаемые в памяти файлы

Большинство операционных систем позволяет пользоваться реализациями виртуальной памяти для отображения (map) файла или только его определенной области в памяти. В таком случае доступ к файлу можно получать так, будто бы он является хранящимся в памяти массивом, что гораздо быстрее, чем получать его посредством традиционных файловых операций.

В конце этого раздела приведена программа, вычисляющая для файла контрольную сумму CRC32 с использованием традиционного файлового ввода и отображаемого в памяти файла. В табл. 1.6 перечислены временные показатели, которые мы получили на одном компьютере при вычислении с помощью этой программы контрольной суммы для занимающего 37 Мбайт файла `rt.jar` из входящего в состав JDK каталога `jre/lib`.

Таблица 1.6. Временные показатели по файловым операциям

Метод	Время
Простой поток входных данных	110 с
Буферизованный поток входных данных	9,9 с
Файл с произвольным доступом	162 с
Отображаемый в памяти файл	7,2 с

В этой таблице видно, что конкретно на данной машине, при отображении в память потребовалось немного меньше времени, чем при применении последовательного ввода с буфером, и гораздо меньше, чем при применении `RandomAccessFile`.

Разумеется, на других машинах эти показатели будут выглядеть несколько иначе, но то, что выигрыш в плане производительности может оказаться значительным при необходимости в применении произвольного доступа, не оставляет никаких сомнений. При потребности в считывании файлов среднего размера, однако, прибегать к отображению в память не имеет никакого смысла.

Пакет `java.nio` делает отображение в памяти (memory mapping) довольно простым процессом. Все требуемые для этого действия перечислены ниже.

Сначала из файла получается *канал* (channel). 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. Получается такой канал путем вызова метода `getChannel`, который был добавлен в класс `FileInputStream`, `FileOutputStream` и `RandomAccessFile`.

```
FileInputStream in = new FileInputStream(. . .);
FileChannel channel = in.getChannel();
```

Далее уже из канала получается объект `MappedByteBuffer` вызовом метода `map` класса `FileChannel`, в котором указывается подлежащая отображению область файла и *режим отображения*. Всего поддерживается три режима отображения, и выглядят они следующим образом.

- `FileChannel.MapMode.READ_ONLY`. Результирующий буфер предусматривает возможность доступа только для чтения. Любые попытки выполнения записи в этот буфер будут приводить к выдаче исключения `ReadOnlyBufferException`.
- `FileChannel.MapMode.READ_WRITE`. Результирующий буфер предусматривает возможность доступа как для чтения, так и для записи, благодаря чему все вно-

симые изменения в определенной точке времени будут записываться обратно в файл. Важно обратить внимание на то, что другие программы, отобразившие тот же самый файл, возможно, сразу этих изменений не увидят. Точное поведение при одновременном отображении файла множеством программ будет зависеть от используемой операционной системы.

- `FileChannel.MapMode.PRIVATE`. Результирующий буфер предусматривает возможность доступа как для чтения, так и для записи, но любые вносимые изменения будут являться собственностью этого буфера и, следовательно, *не* будут передаваться в файл.

Получив требуемый буфер, можно переходить непосредственно к считыванию и записи данных с помощью методов класса `ByteBuffer` и суперкласса `Buffer`.

Буферы поддерживают как последовательный, так и произвольный доступ к данным. У любого буфера имеется *позиция*, передвигаемая посредством операций `get` и `put`. Например, код, необходимый для последовательного прохода по всем байтам в буфере, выглядит следующим образом:

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

А код, необходимый для произвольного доступа, выглядит так:

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

Помимо этого, можно еще также считывать и записывать массивы байтов с помощью следующих методов:

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

И, наконец, с помощью методов

```
getInt
getLong
getShort
getChar
getFloat
getDouble
```

можно считывать значения простых типов, хранящиеся в файле в *двоичном* формате. Как упоминалось ранее, в Java для двоичных данных применяется формат с порядком следования байтов, начиная со старшего (big-endian). При необходимости обработать файл, содержащий двоичные числа в порядке следования байтов, начиная с младшего, однако, можно просто использовать следующий вызов:

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

Для выяснения того, как выглядит текущий порядок следования байтов, можно применить такой вызов:

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



В двух последних методах соглашение об именовании `get/set` не применяется.

Для записи чисел в буфер можно воспользоваться одним из перечисленных ниже методов:

```
putInt
putLong
putShort
putChar
putFloat
putDouble
```

В листинге 1.7 показана программа, вычисляющая для файла контрольную сумму с применением 32-битного циклического кода с избыточностью (32-bit Cyclic Redundancy Checksum — CRC32). Такая контрольная сума часто применяется для определения того, не был ли файл поврежден. Повреждение файла практически неизбежно ведет к изменению этой суммы. В пакете `java.util.zip` содержится класс `CRC32`, вычисляющий такую контрольную сумму для последовательности байтов с помощью следующего цикла:

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



Хорошее описание алгоритма CRC доступно по адресу:
`http://www.relisoft.com/Science/CrcMath.html`.

Детали процесса вычисления CRC не важны. Он демонстрируется здесь просто в качестве примера полезной файловой операции.

Запустить программу можно с помощью следующей команды:

```
java NIOTest имя_файла
```

Листинг 1.7. Содержимое файла `NIOTest.java`

```
1.  import java.io.*;
2.  import java.nio.*;
3.  import java.nio.channels.*;
4.  import java.util.zip.*;
5.
6.  /**
7.   * Эта программа вычисляет контрольную сумму CRC для файла.
8.   * Использование: java NIOTest имя_файла
9.   * @version 1.01 2004-05-11
10.  * @author Cay Horstmann
11.  */
12. public class NIOTest
13. {
14.   public static long checksumInputStream(String filename) throws IOException
15.   {
16.     InputStream in = new FileInputStream(filename);
17.     CRC32 crc = new CRC32();
18.
```

```
19.     int c;
20.     while ((c = in.read()) != -1)
21.       crc.update(c);
22.     return crc.getValue();
23.   }
24.
25.   public static long checksumBufferedInputStream(String filename) throws IOException
26.   {
27.     InputStream in = new BufferedInputStream(new FileInput  Stream(filename));
28.     CRC32 crc = new CRC32();
29.
30.     int c;
31.     while ((c = in.read()) != -1)
32.       crc.update(c);
33.     return crc.getValue();
34.   }
35.
36.   public static long checksumRandomAccessFile(String filename) throws IOException
37.   {
38.    RandomAccessFile file = new RandomAccessFile(filename, "r");
39.     long length = file.length();
40.     CRC32 crc = new CRC32();
41.
42.     for (long p = 0; p < length; p++)
43.     {
44.       file.seek(p);
45.       int c = file.readByte();
46.       crc.update(c);
47.     }
48.     return crc.getValue();
49.   }
50.
51.   public static long checksumMappedFile(String filename) throws IOException
52.   {
53.     FileInputStream in = new FileInputStream(filename);
54.     FileChannel channel = in.getChannel();
55.
56.     CRC32 crc = new CRC32();
57.     int length = (int) channel.size();
58.     MappedByteBuffer buffer = channel.map(FileChannel.MapMode.READ_ONLY, 0, length);
59.
60.     for (int p = 0; p < length; p++)
61.     {
62.       int c = buffer.get(p);
63.       crc.update(c);
64.     }
65.     return crc.getValue();
66.   }
67.
68.   public static void main(String[] args) throws IOException
69.   {
70.     System.out.println("Input Stream:");
71.     long start = System.currentTimeMillis();
72.     long crcValue = checksumInputStream(args[0]);
73.     long end = System.currentTimeMillis();
74.     System.out.println(Long.toHexString(crcValue));
75.     System.out.println((end - start) + " milliseconds");
```

```
76.
77.     System.out.println("Buffered Input Stream:");
78.     start = System.currentTimeMillis();
79.     crcValue = checksumBufferedInputStream(args[0]);
80.     end = System.currentTimeMillis();
81.     System.out.println(Long.toHexString(crcValue));
82.     System.out.println((end - start) + " milliseconds");
83.
84.     System.out.println("Random Access File:");
85.     start = System.currentTimeMillis();
86.     crcValue = checksumRandomAccessFile(args[0]);
87.     end = System.currentTimeMillis();
88.     System.out.println(Long.toHexString(crcValue));
89.     System.out.println((end - start) + " milliseconds");
90.
91.     System.out.println("Mapped File:");
92.     start = System.currentTimeMillis();
93.     crcValue = checksumMappedFile(args[0]);
94.     end = System.currentTimeMillis();
95.     System.out.println(Long.toHexString(crcValue));
96.     System.out.println((end - start) + " milliseconds");
97.   }
98. }
```

API java.io.FileInputStream 1.0

- `FileChannel getChannel()` 1.4

  Возвращает канал для получения доступа к данному потоку.

API java.io.FileOutputStream 1.0

- `FileChannel getChannel()` 1.4

  Возвращает канал для получения доступа к данному потоку.

API java.io.RandomAccessFile 1.0

- `FileChannel getChannel()` 1.4

  Возвращает канал для получения доступа к данному файлу.

API java.nio.channels.FileChannel 1.4

- `MappedByteBuffer map(FileChannel.MapMode mode, long position, long size)`

  Отображает область файла в памяти.

  Параметры:

`mode`	Одна из следующих констант класса `FileChannel.MapMode`: `READ_ONLY`, `READ_WRITE` или `PRIVATE`.
`position`	Начало отображаемой области.
`size`	Размер отображаемой области.

API java.nio.Buffer 1.4

- `boolean hasRemaining()`

  Возвращает `true`, если текущая позиция буфера еще не достигла его предельной позиции.

- `int limit()`

  Возвращает предельную позицию буфера, т.е. первую позицию, в которой больше не доступны никакие значения.

API java.nio.ByteBuffer 1.4

- `byte get()`

  Получает байт из текущей позиции и передвигает текущую позицию на следующий байт.

- `byte get(int index)`

  Получает байт из указанного индекса.

- `ByteBuffer put(byte b)`

  Помещает байт в текущую позицию и передвигает текущую позицию на следующий байт. Возвращает ссылку на данный буфер.

- `ByteBuffer put(int index, byte b)`

  Помещает байт по указанному индексу. Возвращает ссылку на данный буфер.

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

  Заполняет массив байтов или только какую-то область байтового массива с помощью байтов из буфера и передвигает текущую позицию на количество считанных байтов. Если в буфере остается недостаточное количество байтов, тогда ничего не считывает и выдает исключение `BufferUnderflowException`. Возвращает ссылку на данный буфер.

Параметры:	`destination`	Подлежащий заполнению массив байтов.
	`offset`	Смещение подлежащей заполнению области.
	`length`	Длина подлежащей заполнению области.

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

  Помещает все байты из байтового массива или только какой-то его области в буфер и передвигает текущую позицию на количество записанных байтов. Если в буфере слишком много байтов, тогда ничего не записывает и выдает исключение `BufferOverflowException`. Возвращает ссылку на данный буфер.

Параметры:	`source`	Подлежащий записи массив байтов.
	`offset`	Смещение записываемой области.
	`length`	Длина записываемой области.

- *Xxx* `get`*Xxx*`()`
- *Xxx* `get`*Xxx* `(int index)`
- `ByteBuffer put`*Xxx* `(`*xxx* `value)`
- `ByteBuffer put`*Xxx* `(int index,` *xxx* `value)`

  Получает или записывает двоичное число. На месте *Xxx* может быть указано одно из следующих значений: `Int`, `Long`, `Short`, `Char`, `Float` или `Double`.

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

  Устанавливает или получает порядок байтов. В качестве значения для `order` может указываться одна из следующих констант класса `ByteOrder`: `BIG_ENDIAN` или `LITTLE_ENDIAN`.

Структура буферных данных

При использовании подхода с отображением в память создается один единственный буфер, охватывающий либо весь файл целиком, либо только интересующую в нем область. Однако буферы еще также могут применяться для считывания и записи и более скромных фрагментов информации.

В этом разделе дается краткое описание основных операций, которые могут выполняться с объектами `Buffer`. Буфером, в общем, называется массив значений одинакового типа. Класс `Buffer` представляет собой абстрактный класс с такими конкретными подклассами, как `ByteBuffer`, `CharBuffer`, `DoubleBuffer`, `FloatBuffer`, `IntBuffer`, `LongBuffer` и `ShortBuffer`.



Класс `StringBuffer` к этим подклассам буферов никакого отношения не имеет.

На практике чаще всего применяется класс `ByteBuffer` и `CharBuffer`. Как показано на рис. 1.11, каждый буфер обладает:

- *емкостью*, которая никогда не изменяется;
- *позицией*, начиная с которой считывается или записывается следующее значение;
- *пределом*, после которого выполнять считывание или запись не имеет смысла;
- необязательно, *отметкой* для повторения операции считывания или записи.

Все эти значения удовлетворяют следующему условию:

$$0 \le \textit{отметка} \le \textit{позиция} \le \textit{предел} \le \textit{емкость}$$

Буфер работает главным образом по циклу “сначала запись, потом чтение”. Изначально позиция буфера соответствует нулю (0), а предел — емкости. Для добавления значений в буфер нужно вызывать метод `put`. Исчерпав подлежащие добавлению в буфер данные, или заполнив всю его емкость, можно переходить к чтению.

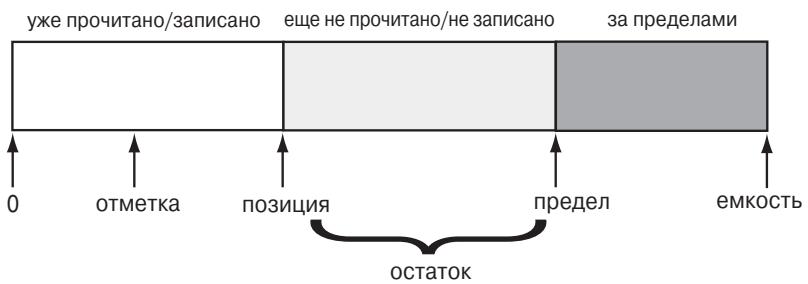


Рис. 1.11. Буфер

Для этого потребуется вызвать метод `flip`, чтобы значение предела стало соответствовать текущей позиции, а позиция — нулю. Далее можно вызывать метод `get` до тех пор, пока метод `remaining` (возвращающий разницу между значением предела и позиции) будет возвращать положительные значения. Считав все значения в буфере, нужно вызвать метод `clear`, чтобы подготовить буфер к следующему циклу записи. Метод `clear` возвращает позицию в нулевое положение и снова приравнивает значение предела к значению емкости.

При желании выполнить повторное считывание данных в буфере, нужно применять такие методы, как `rewind` или `mark/reset`, способы вызова которых более подробно описываются в приведенных ниже врезках API.

API java.nio.Buffer 1.4

- `Buffer clear()`

  Подготавливает данный буфер к записи данных путем приведения его позиции в нулевое положение, а значения предела — к значению емкости. Возвращает `this`.

- `Buffer flip()`

  Подготавливает данный буфер к считыванию данных путем приравнивания предела к позиции, а позиции — к нулю; возвращает `this`.

- `Buffer rewind()`

  Подготавливает данный буфер к повторному считыванию тех же самых значений путем приравнивания позиции к нулю и оставления предела неизменным; возвращает `this`.

- `Buffer mark()`

  Приравнивает отметку данного буфера к позиции; возвращает `this`.

- `Buffer reset()`

  Приравнивает позицию данного буфера к отметке, тем самым позволяя снова выполнять считывание или запись с отмеченной позиции; возвращает `this`.

- `int remaining()`

  Возвращает оставшееся количество доступных для считывания или записи значений, т.е. разницу между пределом и позицией.

- `int position()`

  Возвращает позицию данного буфера.

- `int capacity()`

  Возвращает емкость данного буфера.

API java.nio.CharBuffer 1.4

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

  Считывают либо одно значение `char`, либо диапазон значений `char`, начиная с позиции буфера и передвигая эту позицию по мере считывания символов. Последние два метода возвращают `this`.

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

  Записывают либо одно значение `char`, либо диапазон значений `char`, начиная с позиции буфера и передвигая эту позицию по мере записывания символов. При выполнении считывания из `CharBuffer`, осуществляют считывание всех оставшихся символов. Все методы возвращают `this`.

- `CharBuffer read(CharBuffer destination)`

  Считывает значения из данного буфера и записывает их в пункт назначения до тех пор, пока не будет достигнут его предел. Возвращает `this`.

Блокирование файлов

Возьмем ситуацию, когда множеству одновременно выполняющихся программ требуется изменить один и тот же файл. Очевидно, что необходимо, чтобы эти программы могли как-то общаться между собой, иначе файл может очень легко повредиться.

Средства блокирования файлов позволяют управлять доступом к файлу или диапазону определенных байтов внутри него. Однако в различных операционных системах они выглядят совершенно по-разному, из-за чего, собственного говоря, возможности блокирования файлов и отсутствовали в предыдущих версиях JDK.

Блокирование файлов вовсе не является таким уж распространенным механизмом в прикладных приложениях. Во многих приложениях для хранения данных применяются базы данных, а у баз данных имеются свои механизмы для разрешения проблем с параллельным доступом. Тем, кто сохраняет информацию в плоских (двумерных) файлах и беспокоится о возможности получения к ним параллельного доступа, может быть легче использовать базу данных, чем разрабатывать сложные схемы блокирования файлов.

Тем не менее, бывают ситуации, когда блокирование файлов является крайне необходимым. Например, возьмем приложение, сохраняющее конфигурационный файл с предпочтениями пользователя. В случае вызова пользователем сразу двух экземпляров этого приложения, вполне может случиться так, что обоим из них в какой-то момент времени одновременно понадобится записать какие-то данные в конфигурационный файл. В такой ситуации первый экземпляр должен заблокировать файл. Когда второй экземпляр обнаружит, что файл заблокирован, он может решить, либо подождать, когда файл будет разблокирован, либо просто пропустить процесс записи.

Для блокирования файла может вызываться либо метод `lock`, либо метод `tryLock` класса `FileChannel`:

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

или

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

Первый вызов блокируется и ожидает получения разрешения на блокировку, а второй — сразу же возвращается либо с полученным разрешением на блокировку, либо со значением `null`, если таковое получить не удалось. Файл остается заблокированным до тех пор, пока либо не будет закрыть канал, либо не будет вызван снимающий блокировку метод `release`.

Для блокирования только какой-нибудь определенной части файла может применяться либо вызов:

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

либо же вызов:

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

Указание значение `true` для флага `exclusive` позволяет блокировать доступ к файлу как для чтения, так и для записи, т.е. организовать блокировку с обеспечением монопольного доступа, а указание для него значения `false` позволяет организовать блокировку с обеспечением *совместного доступа* (shared lock), при которой множеству процессов разрешено выполнять считывание данных из файла, но не разрешено устанавливать блокировку с обеспечением монопольного доступа. Блокировки с обеспечением совместного доступа поддерживают не все операционные системы. Из-за чего не исключено получение блокировки с обеспечением монопольного доступа (exclusive lock) даже в тех случаях, когда была запрошена блокировка с обеспечением совместного доступа. Для выяснения того, о каком виде блокировки идет речь, нужно вызывать метод `isShared` класса `FileLock`.



В случае блокирования заключительной части файла и последующего разрастания файла за пределы заблокированной области, дополнительная область блокироваться не будет. Для блокирования всех байтов нужно указать для размера значение `Long.MAX_VALUE`.

Кроме того, еще важно не забывать о том, что поведение средств блокирования файлов зависит от используемой системы. Ниже перечислены некоторые моменты, на которые не помешает обращать внимание.

- На некоторых системах блокирование файлов является лишь *желательным* (а не обязательным). При невозможности получить разрешение на блокировку, приложение может все равно начинать выполнять запись файл, даже если доступ к этому файлу сейчас заблокирован другим приложением.
- В некоторых системах нельзя одновременно блокировать файл и отображать его в памяти.
- Блокировки файлов удерживаются всей виртуальной машиной Java. В случае запуска с помощью одной и той же виртуальной машиной сразу двух программ (вроде аплета или программы запуска приложений), эти программы не смогут по отдельности получать разрешение на блокировку одного и того же файла. В случае если виртуальная машина уже удерживает другую перекрывающую блокировку для того же самого файла, методы `lock` и `tryLock` будут просто выдавать исключение `OverlappingFileLockException`.
- В некоторых системах закрытие канала приводит к снятию с лежащего в основе файла всех блокировок, которые удерживаются виртуальной машиной Java. Из-за этого лучше избегать множества каналов на одном и том же заблокированном файле.
- Особенности блокирования файлов в файловой системе, подключенной к сети, очень сильно зависят от используемой системы, из-за чего такого подхода, пожалуй, лучше стараться избегать.

API java.nio.channels.FileChannel 1.4

- `FileLock lock()`

  Получает разрешение на блокировку, исключающую всякий доступ ко всему файлу. Блокирует доступ до самых тех пор, пока это разрешение не будет получено.

- `FileLock tryLock()`

  Получает разрешение на блокировку, исключающую всякий доступ ко всему файлу, или возвращает `null`, если получить такое разрешение не удается.

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

  Получает разрешение на блокировку доступа к определенной области файла. Первый метод блокирует доступ до самых тех пор, пока такое разрешение не будет получено, а второй — возвращает `null`, если получить это разрешение не удается.

  Параметры:

`position`	Начало подлежащей блокированию области.
`size`	Размер подлежащей блокированию области
`shared`	`true`, если требуется блокировка с обеспечением совместного доступа (shared lock), и `false` — если блокировка с обеспечением монопольного доступа.

API java.nio.channels.FileLock 1.4

- `void release()`

  Снимает блокировку.

Регулярные выражения

Регулярные выражения применяются для указания шаблонов строк. Их можно использовать всякий раз, когда требуется отыскать строки, соответствующие конкретному шаблону. Например, одна из наших демонстрационных программ умеет отыскивать в HTML-файле все гиперссылки путем поиска строк, соответствующих следующему шаблону: `<a href="...">`.

Конечно, для определения шаблона обозначение `...` является не достаточно точным. Необходимо как можно конкретнее указывать, какая именно последовательность символов должна считаться настоящим совпадением. Этот требует применения при каждом описании шаблона специального синтаксиса.

Ниже приведен простой пример. Регулярное выражение

```
[Jj]ava.+
```

указывает считать совпадением любую строку, которая отвечает следующим условиям:

- имеет в начале букву `J` или `j`;
- в качестве трех следующих букв содержит буквы `ava`;
- в остальной части состоит из одного или более других (произвольных) символов.

Например, строка `"javanese"` будет являться соответствующим данному регулярному выражению совпадением, а строка `"Core Java"` — нет.

Как видите, для того чтобы понять смысл регулярного выражения, нужно хотя бы немного разбираться в синтаксисе. К счастью, для большинства целей вполне хватает небольшого набора довольно простых конструкций.

- *Класс символов* (character class) — набор вариантов символов, заключенный в квадратные скобки, например: `[Jj]`, `[0-9]`, `[A-Za-z]` или `[^0-9]`. Здесь символом - обозначается диапазон (все символы, значение Unicode которых вписывается в указанные рамки), а символом ^ — дополнение (все символы кроме указанных).
- Существует много предопределенных классов символов, вроде класса `\d` (для цифр) или класса `\p{Sc}` (для валютного символа Unicode). См. табл. 1.7 и 1.8.
- Многие символы указывают считать совпадением самих себя, подобно `ava` в предыдущем примере.
- Символ . указывает считать совпадением любой символ (кроме, возможно, символов окончания строки, что зависит от параметров флагов).
- Символ \ служит символом отмены, например, комбинация `\.` означает, что совпадением должна считаться точка, а комбинация `\\`, соответственно — что совпадением должен считаться символ обратной косой черты.
- Символы ^ и $ означают, что совпадением должны считаться, соответственно, начало и конец строки.
- Если *X* и *Y* являются регулярными выражениями, тогда *XY* означает, что совпадением должно считаться "любое совпадение для *X*, за которым следует совпадение для *Y*", а *X* | *Y*— что совпадением должно считаться "любое совпадение для *X* или *Y*".
- К выражению *X* можно применять *квантификаторы* (quantifiers), наподобие *X*+ (1 или больше), *X** (0 или больше) и *X*? (0 или 1).
- По умолчанию квантификатор указывает, что совпадением должно считаться наибольшее возможное количество повторений, делающее весь поиск соответствий в общем успешным. Это поведение можно менять с помощью суффикса `?` (указывающего на необходимость принятия даже минимального или скудного совпадения т.е. считать совпадением даже наименьшее число повторений) и суффикса + (указывающего на необходимость принятия только максимального или наиболее точного совпадения, т.е. считать совпадением только наибольшее число повторений, даже если это чревато неудачным завершением всего процесса поиска соответствий).

  Например, строка `cab` будет считаться совпадением в случае выражения `[a-z]*ab`, но не в случае выражения `[a-z]*+ab`. В первом случае из-за конструкции `[a-z]*` совпадением будет признаваться лишь символ `c`, благодаря чему символы `ab` смогут подбираться уже по остальной части шаблона. А во втором случае из-за более строгой версии `[a-z]*+` совпадением сразу будет признаваться только комбинация символов `cab`, вследствие чего, соответственно, подбор соответствий для остальной части шаблона проводиться вообще не будет.
- Для определения подвыражений можно использовать *группы*. Все группы нужно обязательно заключать в круглые скобки, например: `([+-]?)([0-9]+)`. После этого можно просить обнаружителя совпадений возвращать совпадения либо для каждой группы, либо только для определенной группы путем использования конструкции `\n`, в которой на месте `n` следует указывать номер нужной группы (начиная с `\1`).

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 является полностью стандартизированным и может выглядеть по-разному в разных применяющих их программах и библиотеках. Хотя общее соглашение по базовым конструкциям, конечно, существует, имеется масса очень раздражающих отличий в деталях. В классах регулярных выражений в Java применяется синтаксис, подобный, но все-таки не полностью совпадающий с тем, что применяется в языке Perl. В табл. 1.7 перечислены все конструкции синтаксиса Java. Более подробную информацию о синтаксисе регулярных выражения можно найти либо в документации по API для класса `Pattern`, либо в книге Джефри Фридла (Jeffrey E. F. Friedl) под названием *Mastering Regular Expressions*(O'Reilly and Associates, 1997 г.).

Самым простым способом применения регулярного выражения является выполнение проверки на предмет того, соответствует ли его требованиям конкретная строка. Ниже приведена программа, выполняющая такую проверку в Java. В этой программе первым делом из представляющей регулярное выражение строки создается объект `Pattern`. Далее из этого объекта получается объект `Matcher` и вызывается его метод `matches`:

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

Таблица 1.7. Синтаксис регулярных выражений

Синтаксис	Описание
Символы	
`c`	Символ *c*.
`\u`*nnnn*, `\x`*nn*, `\0`*n*, `\0`*nn*, `\0`*nnn*	Блок кода с заданным шестнадцатеричным или восьмеричным значением.
`\t, \n, \r, \f, \a, \e`	Управляющие символы табуляции, новой строки, возврата каретки, перевода страницы, предупреждения и перехода.
`\c`*c*	Управляющий символ, соответствующий символу *c*.
Классы символов	
`[`C_1C_2`...]`	Любой из символов, представленных C_1, C_2 ... С помощью C_i могут указываться 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 в случае установки флага `DOTALL`).
`\d`	Цифровой символ `[0-9]`.
`\D`	Не цифровой символ `[^0-9]`.
`\s`	Пробельный символ `[ \t\n\r\f\x0B]`.
`\S`	Не пробельный символ.

Окончание табл. 1.7

Синтаксис	Описание
`\w`	Словесный символ `[a-zA-Z0-9_]`.
`\W`	Не словесный символ.
`\p{`*name*`}`	Именованный класс символов.
`\P{`*name*`}`	Дополнение к именованному классу символов.
Обнаружители граничных совпадений	
`^ $`	Начало и конец ввода (или начало и конец строки в многострочном режиме).
`\b`	Словесная граница.
`\B`	Не словесная граница.
`\A`	Начало ввода.
`\z`	Конец ввода.
`\Z`	Конец ввода без учета последнего символа завершения строки.
`\G`	Конец предыдущего совпадения.
Квантификаторы	
`X?`	Необязательное наличие X.
`X*`	Наличие X 0 или более раз.
`X+`	Наличие X 1 или более раз.
`X{`*n*`} X{`*n*`,} X{`*n*`,`*m*`}`	Наличие X *n*-е количество раз, как минимум *n*-е количество раз, между *n*-м и *m*-м количеством раз.
Суффиксы квантификаторов	
`?`	Указывает считать совпадением даже самое скудное или минимальное соответствие.
`+`	Указывает считать совпадением только максимальное или наиболее точное соответствие.
Операции над множествами	
`XY`	Любая строка из *X*, за которой следует любая строка из *Y*.
`X|Y`	Любая строка из *X* или *Y*.
Группирование	
`(X)`	Фиксация соответствующей *X* строки как группы.
`\`*n*	Совпадение *n*-й группы.
Символы перехода	
`\`*c*	Символ *c* (ни в коем случае не должен представлять собой алфавитный символ).
`\Q . . . \E`	Цитирование... дословная передача.
`(? . . . )`	Специальная конструкция; см. врезки API, посвященные классу `Pattern`.

Таблица 1.8. Имена предопределенных классов символов

Имя класса символов	Объяснение
`Lower`	ASCII-символы `[a-z]` в нижнем регистре.
`Upper`	ASCII-символы `[A-Z]` в верхнем регистре.
`Alpha`	ASCII-буквы алфавита `[A-Za-z]`.
`Digit`	ASCII-цифры `[0-9]`.
`Alnum`	ASCII-буквы алфавита или цифры `[A-Za-z0-9]`.
`XDigit`	Шестнадцатеричные цифры `[0-9A-Fa-f]`.
`Print` или `Graph`	Печатаемые ASCII-символы `[\x21-\x7E]`.
`Punct`	Небуквенные ASCII-символы или цифры `[\p{Print}&&\P{Alnum}]`.
`ASCII`	Все ASCII-символы `[\x00-\x7F]`.
`Cntrl`	Управляющие ASCII-символы `[\x00-\x1F]`.
`Blank`	Символы пробела или табуляции `[ \t]`.
`Space`	Пробельные символы `[ \t\n\r\f\0x0B]`.
`javaLowerCase`	Символы в нижнем регистре, определяемые методом `Character.isLowerCase()`.
`javaUpperCase`	Символы в верхнем регистре, определяемые методом `Character.isUpperCase()`.
`javaWhitespace`	Пробельные символы, определяемые методом `Character.isWhitespace()`.
`javaMirrored`	Зеркальные символы, определяемые методом `Character.isMirrored()`.
`In`*`Block`*	Под *`Block`* подразумевается имя блока Unicode-символов без пробелов, вроде `BasicLatin` и `Mongolian`. Полный перечень имен таких блоков можно найти по адресу `http://www.unicode.org`.
`Category` или `In`*`Category`*	Под *`Category`* подразумевают имя категории символов Unicode, вроде `L` (буква) или `Sc` (символ валюты). Полный перечень имен этих категорий можно найти по адресу `http://www.unicode.org`.

В роли входных данных для обнаружителя совпадений может выступать объект любого класса, который реализует интерфейс `CharSequence`, например, `String`, `StringBuilder` или `CharBuffer`.

При компиляции шаблона можно устанавливать один и более флагов, например:

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

Всего поддерживается шесть флагов, которые перечислены ниже.

- `CASE_INSENSITIVE`. Указывает отыскивать совпадения среди символов без учета регистра. По умолчанию данный флаг принимает во внимание только символы US ASCII.
- `UNICODE_CASE`. При использовании в сочетании с флагом `CASE_INSENSITIVE`, указывает применять для поиска совпадений регистр букв Unicode.
- `MULTILINE`. `^` и `$` указывают сопоставлять начало и конец строки, а не все входные данные.

- UNIX_LINES. Указывает при поиске соответствий для ^ и $ в многострочном режиме, распознавать ли в качестве символа конца строки только символ '\n'.
- DOTALL. Указывает, что для символа . совпадениями должны считаться все символы, и символы конца строки включительно.
- CANON_EQ. Указывает, брать ли в расчет и каноническую эквивалентность символов Unicode. Например, символ u, за которым следует символ ¨ (трема — диакритический знак в виде двух горизонтальных точек над гласной), будет соответствовать символу ü.

Если регулярное выражение содержит группы, тогда объект Matcher может сообщать их границы. Методы

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

выдают начальный и конечный индекс конкретной группы.

Извлечь совпавшую строку можно и просто путем выполнения следующего вызова:

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

Под группой 0 подразумеваются все входные данные; индексом первой фактической группы является 1. Для получения информации об общем количестве групп необходимо вызывать метод groupCount.

Вложенные группы упорядочиваются путем открытия круглых скобок. Например, в случае использования такого шаблона:

```
((1?[0-9]):([0-5][0-9]))[ap]m
```

и следующих входных данных:

```
11:59am
```

обнаружитель совпадений сообщит о таких 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 1.8 приведен код программы, которая сначала предлагает указать шаблон, а затем — подлежащие сопоставлению с ним строки, после чего сообщает о том, совпадают введенные строки с шаблоном или нет. Если в совпавших введенных строках и шаблоне присутствуют группы, тогда эта программа распечатывает границы групп с использованием круглых скобок, например:

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

Листинг 1.8. Содержимое файла RegexTest.java

```
1.  import java.util.*;
2.  import java.util.regex.*;
3.  /**
4.   * Эта программа позволяет сопоставлять регулярные выражения.
5.   * Для выполнения сопоставления нужно ввести шаблон и подлежащие
6.   * сопоставлению с ним строки, а для выхода — пустую строку.
```

```
7.    * Если в шаблоне присутствуют группы, в совпадении отображаются
8.    * границы этих групп.
9.    * @version 1.01 2004-05-11
10.   * @author Cay Horstmann
11.   */
12.  public class RegExTest
13.  {
14.    public static void main(String[] args)
15.    {
16.      Scanner in = new Scanner(System.in);
17.      System.out.println("Enter pattern: ");
18.      String patternString = in.nextLine();
19.
20.      Pattern pattern = null;
21.      try
22.      {
23.        pattern = Pattern.compile(patternString);
24.      }
25.      catch (PatternSyntaxException e)
26.      {
27.        System.out.println("Pattern syntax error");
28.                        // Ошибка в синтаксисе шаблона
29.        System.exit(1);
30.      }
31.
32.      while (true)
33.      {
34.        System.out.println("Enter string to match: ");
35.                        // Введите подлежащую сопоставлению строку
36.        String input = in.nextLine();
37.        if (input == null || input.equals("")) return;
38.        Matcher matcher = pattern.matcher(input);
39.        if (matcher.matches())
40.        {
41.          System.out.println("Match"); // Совпадает
42.          int g = matcher.groupCount();
43.          if (g > 0)
44.          {
45.            for (int i = 0; i < input.length(); i++)
46.            {
47.              for (int j = 1; j <= g; j++)
48.               if (i == matcher.start(j))
49.                 System.out.print('(');
50.              System.out.print(input.charAt(i));
51.              for (int j = 1; j <= g; j++)
52.               if (i + 1 == matcher.end(j))
53.                 System.out.print(')');
54.            }
55.            System.out.println();
56.          }
57.        }
58.       else
59.          System.out.println("No match");
60.                          // Совпадений нет
61.     }
62.    }
63.  }
```

Обычно требуется не сопоставлять все входные данные с регулярным выражением, а просто отыскивать в них одну или более совпадающих строк. Для поиска следующего совпадения служит метод `find` класса `Matcher`. Если он возвращает `true`, тогда дальше можно использовать методы `start` и `end` для выяснения масштабов совпадения.

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

В листинге 1.9 показана программа, приводящая этот механизм в действие. Она отыскивает на Web-странице все гипертекстовые ссылки и отображает их на экране. Чтобы запустить эту программу, нужно указать в командной строке какой-нибудь URL-адрес, например такой:

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

Листинг 1.9. Содержимое файла `HrefMatch.java`

```
1.  import java.io.*;
2.  import java.net.*;
3.  import java.util.regex.*;
4.
5.  /**
6.   * Эта программа отображает все имеющиеся на Web-странице URL-адреса
7.   * путем отыскивания совпадений для регулярного выражения, которое
8.   * описывает HTML-дескриптор <a href=...>. Запускается она
9.   * следующим образом:
10.  * java HrefMatch URL
11.  * @version 1.01 2004-06-04
12.  * @author Cay Horstmann
13.  */
14. public class HrefMatch
15. {
16.   public static void main(String[] args)
17.   {
18.     try
19.     {
20.     // Извлечение строки URL из командной строки
21.     // или использование стандартного URL
22.     String urlString;
23.     if (args.length > 0) urlString = args[0];
24.     else urlString = "http://java.sun.com";
25.
26.     // открытие считывателя для URL
27.     InputStreamReader in = new InputStreamReader(new URL(urlString).openStream());
28.
29.     // считывание содержимого в строковый конструктор
30.     StringBuilder input = new StringBuilder();
31.     int ch;
32.     while ((ch = in.read()) != -1)
33.     input.append((char) ch);
34.
35.     // поиск всех вхождений шаблона
36.     String patternString = "<a\\s+href\\s*=\\s*(\"[^\"]*\"|[^\\s>])\\s*>";
```

```
37.     Pattern pattern = Pattern.compile(patternString, Pattern.CASE_INSENSITIVE);
38.     Matcher matcher = pattern.matcher(input);
39.
40.     while (matcher.find())
41.     {
42.       int start = matcher.start();
43.       int end = matcher.end();
44.       String match = input.substring(start, end);
45.       System.out.println(match);
46.     }
47.    }
48.    catch (IOException e)
49.    {
50.      e.printStackTrace();
51.    }
52.    catch (PatternSyntaxException e)
53.    {
54.      e.printStackTrace();
55.     }
56.   }
57.  }
```

Метод `replaceAll` класса `Matcher` заменяет все вхождения регулярного выражения указанными в строке замены символами. Например, согласно приведенным ниже инструкциям все последовательности цифр должны заменяться символом #. `Pattern pattern = Pattern.compile("[0-9]+");`

```
Matcher matcher = pattern.matcher(input);
String output = matcher.replaceAll("#");
```

В строке замены могут содержаться ссылки на присутствующие в шаблоне группы: `$n` заменяется n-й группой. Для включения в текст замены символа `$` нужно использовать комбинацию `\$`.

Метод `replaceFirst` заменяет только первое вхождение шаблона.

И, наконец, у класса `Pattern` еще также имеется метод `split`, который разбивает входные данные на массив строк, используя в качестве границ совпадения с регулярным выражением. Например, в приведенных ниже инструкциях указывается разбивать входные данные на лексемы, используя в качестве разделителей окруженные (необязательно) пробелами знаки пунктуации.

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

API java.util.regex.Pattern 1.4

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

  Компилирует строку регулярного выражения 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 `matcher`, который можно использовать для обнаружения совпадений с шаблоном во входных данных.

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

  Разбивает строку входных данных на маркеры, причем вид разделителей определяет шаблон. Возвращает массив маркеров. Разделители не являются частью маркеров.

  Параметры: `input` Строка, подлежащая разбиению на маркеры.

  `limit` Максимальное количество строк, которое нужно получить. В случае указания `limit - 1`, в последнюю запись возвращаемого массива помещаются оставшиеся неразбитые входные данные, в случае указания `limit ≤ 0`, разбивке подвергаются все входные данные, а в случае указания `limit 0`, замыкающие пустые строки в возвращаемый массив не добавляются.

API java.util.regex.Matcher 1.4

- `boolean matches()`

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

- `boolean lookingAt()`

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

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

  Пытается отыскать следующее совпадение и если ему это удается, возвращает `true`.

  Параметры: `start` Индекс, с которого подлежит начинать поиск.

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

  Возвращает начальную или конечную позицию текущего совпадения.

- `String group()`

  Возвращает текущее совпадение.

- `int groupCount()`

  Возвращает информацию о количестве групп во входном шаблоне.

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

  Возвращает 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 строку, соответствующую заданной группе.

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

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

  Возвращает строку, получаемую из входных данных `Matcher` путем замены всех или только первого совпадения строкой замены.

  Параметры: `replacement` Строка замены. Может содержать ссылки на группу шаблона в виде `$`*n*. Добавлять символ `$` нужно следующим образом: `\$`.

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

  Сбрасывает состояние `Matcher`. Второй метод заставляет `Matcher` обрабатывать другие входные данные. Оба эти метода возвращают `this`.

Теперь читатели увидели, как можно выполнять операции ввода и вывода в Java, а также вкратце ознакомились с пакетом регулярных выражений, входящим в состав спецификации “новых средств для обработки ввода-вывода”. В следующей главе речь пойдет об операциях обработки данных XML.

Глава 2

ЯЗЫК XML

В ЭТОЙ ГЛАВЕ...

- Общие сведения об XML
- Разбор XML-документа
- Проверка корректности документа
- Поиск информации с помощью XPath
- Использование пространств имен
- Потоковые анализаторы
- Создание XML-документов
- XSLT-преобразования

В предисловии к книге Дона Бокса (Don Box) и др. *Essential XML* (Addison-Wesley Professional, 2001 г.) говорится, что "...язык XML пришел на смену языку Java, шаблонам проектирования и объектной технологии...". Это, конечно же, шутка. Действительно, XML очень удобно использовать для описания и представления структурированной информации, однако этот язык не является универсальным средством на все случаи жизни, и для его эффективного использования потребуются также специализированные промышленные стандарты и библиотеки. Более того, XML совсем не заменяет, а всего лишь дополняет Java. В конце 1990-х гг. компании IBM, Apache и многие другие приступили к созданию Java-библиотек для обработки XML-данных. Начиная с пакета Java SE 1.4, компания Sun интегрировала наиболее важные библиотеки в состав платформы Java.

В этой главе описаны основы языка XML, а также ключевые инструменты для работы с XML-данными, содержащиеся в библиотеке Java. Как и прежде, здесь рассматриваются случаи, когда использование XML было бы вполне обоснованным, а также ситуации, в которых вполне можно обойтись и без данного языка, используя другие проверенные временем способы проектирования и программирования.

Введение в XML

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

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

Для считывания такого файла с помощью одного метода нужно применить класс `Properties`. Однако этот прекрасный инструмент 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 параметре; это в большей степени соответствовало бы объектно-ориентированному подходу:

```
font=Times Roman 12
```

Но для разбора такого выражения потребуется довольно громоздкий код, так как необходимо определить, где заканчивается название шрифта и начинается его размер.

Дело в том, что файлы свойств имеют так называемую плоскую (двумерную) иерархию. Иногда программисты предпринимают попытки обойти данное ограничение с помощью составных имен.

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

Еще один недостаток формата файла свойств состоит в том, что имена параметров должны быть уникальными. Для хранения последовательности значений придется создать имена наподобие приведенных ниже.

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

Язык XML позволяет решить эти проблемы. Он создан для представления иерархических структур и более гибок по сравнению с файлом свойств.

Например, XML-файл с конфигурационными параметрами программы может выглядеть так, как показано ниже.

```
<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 каких-либо затруднений выражать любую иерархическую структуру с повторяющимися элементами.

Как видите, XML-файл имеет очень ясную структуру, напоминающую структуру HTML-файла. Дело в том, что оба эти языка созданы на базе стандартного обобщенного языка разметки SGML (Standard Generilized Markup Language).

Язык SGML с 1970-х гг. 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. В основном проблемы с данным языком 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 применением макросов и клавиш быстрого доступа. Язык XML представляет собой упрощенную версию SGML. Как это часто бывает в жизни, все гениальное просто, поэтому XML сразу же и с энтузиазмом был воспринят теми специалистами, которые многие годы избегали работать с SGML.



Очень удачно составленное описание стандарта языка XML можно найти по адресу: `http://www.xml.com/axml/axml/html`.

Несмотря на общие корни, языки XML и HTML имеют ряд существенных различий.

- В отличие от HTML, в языке XML учитывается регистр символов, поэтому дескрипторы <H1> и <h1> в XML считаются разными.
- В языке HTML некоторые закрывающие дескрипторы могут отсутствовать. Например, разработчик документа может пропустить </p> или , если из контекста ясно, где заканчивается абзац или пункт списка. В XML это не разрешается.
- Для элементов, тело которых отсутствует, в XML предусмотрена сокращенная запись открывающего дескриптора, совмещенного с закрывающим. В этом случае открывающий дескриптор заканчивается символом /, например . Это означает, что наличие закрывающего дескриптора </img> подразумевается по умолчанию.
- В XML значения атрибутов должны быть помещены 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. В языке XML для значений атрибутов width и height нужно указать кавычки, т.е. width="300" и height="300".
- В HTML допускается использование имен атрибутов без указания их значений, например, <input type="radio" name="language" value="Java" **checked**>. В XML все атрибуты должны иметь значения, например, checked="true" или checked="checked".



В настоящее время при создании Web-документов рекомендуется использовать стандарт XHTML, который упрощает переход от HTML к XML. Описание XHTML можно найти по адресу http://www.w3c.org/TR/xhtml1/. Стандарт XHTML поддерживается всеми современными Web-браузерами, но не всеми инструментальными средствами, предназначенными для создания HTML-документов, формируют код, не совместимый с XHTML. Поскольку популярность XHTML растет, описанные в этой главе XML-инструменты можно применять для анализа Web-документов.

Структура XML-документа

XML-документ начинается со следующего заголовка:

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

Заголовок может выглядеть и так, как показано ниже:

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

Строго говоря, заголовок использовать не обязательно, но все же настоятельно рекомендуется включать его в состав документа.



Язык SGML был предназначен для обработки документов, поэтому XML-файлы принято называть *документами*, хотя многие XML-файлы описывают такие наборы данных, для которых этот термин плохо подходит.

После заголовка обычно располагается *определение типа документа*, или *DTD* (document type definition), пример которого приведен 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 главе.

Наконец, тело XML-документа содержит *корневой элемент* (root element), в состав которого могут входить другие элементы:

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

Каждый элемент может содержать *дочерние элементы* (child element), текст либо и то, и другое. В приведенном выше примере в состав элемента `font` входят два дочерних элемента: `name` и `size`. Элемент `name` содержит текст `"Helvetica"`.



XML-документы рекомендуется формировать так, чтобы элементы содержали *либо* дочерние элементы, *либо* текст. Иначе говоря, следует избегать создания структур, подобных показанной ниже.

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

Такая структура в спецификации XML называется *смешанным содержимым* (mixed contents). Как показано дальше, если не применять смешанное содержимое, вы упростите процедуру разбора документа.

XML-элементы могут содержать атрибуты:

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

Среди разработчиков XML-документов нет единого мнения о том, какие данные следует представлять с помощью элементов, а какие посредством атрибутов.

Рассмотрим строку с описанием шрифта:

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

На первый взгляд строка с атрибутами выглядит гораздо проще, чем показанная ниже иерархия элементов.

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

Однако атрибуты гораздо менее гибкие. Предположим, что в определение размера шрифта нужно добавить единицу измерения. При использовании атрибутов единицы измерения следует указать в значении атрибута:

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

Теперь придется также позаботиться о создании дополнительного кода для синтаксического анализа строки "36 pt", а именно этого стремились избежать создатели языка XML. Более элегантным решением было бы применение атрибута в элементе size.

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

Широко распространенное правило гласит, что атрибуты следует использовать не для указания значения, а только при изменении его интерпретации. Если возникает ситуация, в которой невозможно понять, является ли какой-то атрибут изменением интерпретации значения либо нет. В таком случае следует отказаться от атрибута и применять элемент. Во многих полезных XML-документах атрибуты вовсе не предусмотрены.



В языке HTML имеется очень простое правило: если данные не отображаются на Web-странице, значит, это атрибут. Рассмотрим следующую гипертекстовую ссылку:

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

Строка Java Technology отображается на Web-странице, но URL этой гипертекстовой ссылки не выводится.

Это правило не всегда подходит для XML-данных, поскольку информация в XML-файле не всегда предназначена непосредственно для пользователей.

Элементы и текст являются основными компонентами XML-документов, но можно также встретить некоторые другие инструкции.

- *Ссылки на символы* имеют вид &#*десятичное_значение*; или &#x*шестнадцатеричное_значение*;. Например, ссылки é или Ù обозначают символ é.
- *Ссылки на примитивы* имеют вид &*имя*;. Например, перечисленные ниже ссылки на примитивы имеют предопределенные значения и соответствуют символам <, >, &, " и '. В DTD можно также указать другие ссылки на примитивы:

  ```
  &lt;
  &gt;
  &amp;
  &quot;
  &apos;
  ```

- *CDATA-разделы* ограничены группами символов <![CDATA[и]]>. Они предназначены для включения строк с символами < > &, которые не следует интерпретировать как символы разметки.

  ```
  <![CDATA[< & > are my favorite delimiters]]>
  ```

  В CDATA-разделах не допускается использование строки типа]]>, поэтому их нужно применять очень осторожно. Часто они выполняют функции “лазейки” для вставки в XML-документ существующих данных.

- *Инструкции обработки* представляют собой инструкции для приложений, обрабатывающих XML-документы. Инструкции ограничиваются символами <? и ?>.

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

  Каждый XML-документ начинается со следующей инструкции обработки:

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

- *Комментарии* ограничиваются символами <!-- и -->.

  ```
  <!-- Текст комментария -->
  ```

  В комментариях не допускается использование строки типа --. Комментарии предназначены для пользователей, а потому не следует помещать в них скрытые команды. Для выполнения команд предназначены инструкции обработки.

Разбор XML-документа

Для обработки XML-документа необходимо выполнить *синтаксический анализ*, или *разбор* (parse) его содержимого. *Анализатором* (parser) называется программа, которая считывает файл, подтверждает корректность его формата, разбивает данные на составные элементы и предоставляет программисту доступ к ним. Ниже приведены два основных типа XML-анализаторов.

- Древовидные анализаторы (например, DOM-анализатор — Document Object Model), которые считывают XML-документ и представляют его в виде древовидной структуры.
- Потоковые анализаторы (например, SAX-анализатор — Simple API for XML), которые генерируют события по мере чтения XML-документа.

DOM-анализатор проще в использовании, поэтому сначала рассматривается именно он. Потоковый анализатор обычно применяется для обработки длинных документов, когда для древовидного представления XML-данных требуется большой объем памяти. Кроме того, его можно использовать для извлечения отдельных элементов XML-документа без учета контекста. Более подробную информацию об этом можно прочитать в разделе "Потоковые анализаторы".

Интерфейс DOM-анализатора стандартизован консорциумом W3C. Пакет `org.w3c.dom` содержит определения типов интерфейсов, например, `Document` и `Element`. Ряд компаний, в частности IBM и Apache, разработали собственные варианты DOM-анализаторов, которые реализуют эти интерфейсы. В API обработки XML-документов (Java API for XML Processing — JAXP) компании Sun предусмотрена возможность подключения таких анализаторов. Кроме того, компания Sun включила в состав пакета JDK собственный анализатор, который упоминается далее в главе.

Для считывания XML-документа потребуется объект `DocumentBuilder`, который можно получить с помощью класса `DocumentBuilderFactory`:

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

Теперь можно приступать к чтению данных из файла:

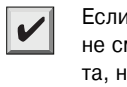
```
File f = ...;
Document doc = builder.parse(f);
```

Для этой цели также можно использовать URL:

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

Строго говоря, чтение XML-данных можно организовать из любого входного потока:

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



Если в качестве источника данных используется произвольный входной поток, анализатор не сможет найти те файлы, расположение которых указано относительно данного документа, например, DTD-файл, находящийся в том же каталоге. Для решения этой проблемы потребуется установить “определитель примитивов” (entity resolver).

Объект `Document` является внутренним представлением древовидной структуры XML-документа. Он состоит из экземпляров классов, реализующих интерфейс `Node` и его дочерние интерфейсы. На рис. 2.1 показана иерархия интерфейсов.

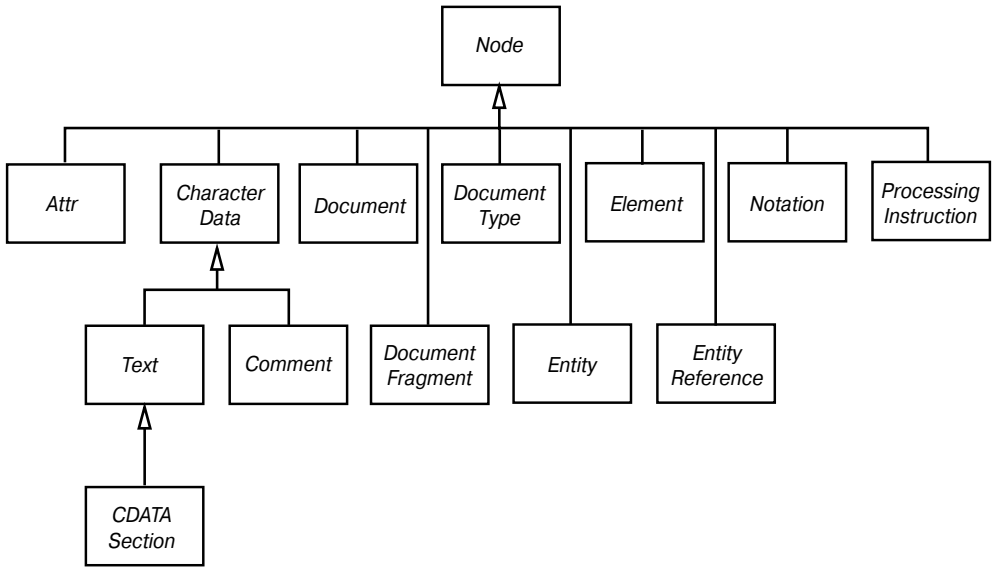


Рис. 2.1. Интерфейс `Node` и его дочерние интерфейсы

Анализ содержимого документа начинается с вызова метода `getDocumentElement()`, который возвращает корневой элемент:

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

Предположим, что обрабатывается следующий XML-документ:

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

В результате вызова метода `getDocumentElement()` будет получен элемент `font`.

Метод `getTagName()` возвращает имя дескриптора элемента. Так, в приведенном выше примере в результате вызова метода `root.getTagName()` будет получена строка `"font"`.

Для извлечения элементов, дочерних по отношению к данному (это могут быть вложенные элементы, текст, комментарии или другие узлы), предназначен метод `getChildNodes()`, который возвращает набор данных типа `NodeList`. Этот тип существовал еще до создания библиотеки наборов данных Java, а потому для работы с ним используется специальный протокол. Метод `item()` возвращает элемент набора с заданным индексом, а метод `getLength()` — общее количество элементов. Таким образом, для перечисления всех дочерних элементов можно применить следующий код:

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

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

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

Однако анализатор сообщит, что имеется пять дочерних элементов:

- разделитель (т.е. символ конца строки, отступ или пробел) между `<font>` и `<name>`;
- элемент `name`;
- разделитель между `</name>` и `<size>`;
- элемент `size`;
- разделитель между `</size>` и `</font>`.

На рис. 2.2 показано дерево DOM.

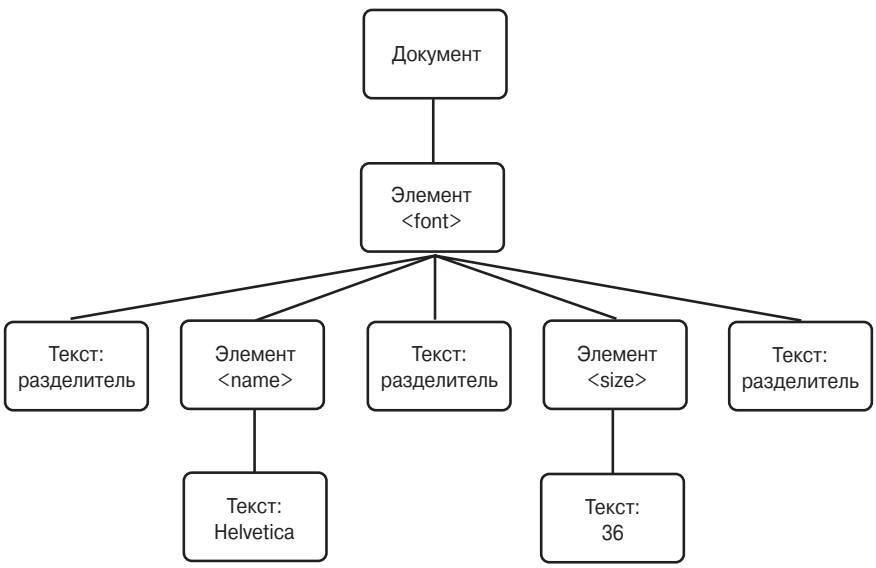


Рис. 2.2. Простое дерево DOM

Если требуется обработать только вложенные элементы, нужно игнорировать все разделители. Это можно сделать с помощью приведенного ниже кода.

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

```
   if (child instanceof Element)
   {
      Element childElement = (Element)child;
      ...
   }
}
```

Теперь будут выявлены только элементы `name` и `size`.

Прочитав следующий раздел, вы увидите, что данную задачу можно решить еще лучше, если воспользоваться DTD. В таком случае анализатор имеет информацию о том, предусмотрены ли в качестве дочерних элементов текстовые узлы, поэтому игнорирует пробелы, символы табуляции и другие разделители подобного рода.

При анализе элементов `name` и `size` нужно извлечь содержащиеся в них текстовые строки, которые находятся в дочерних узлах типа `Text`. Поскольку известно, что других дочерних узлов нет, можно использовать метод `getFirstChild()` без перебора содержимого набора данных `NodeList`. После этого с помощью метода `getData()` можно извлечь строку из узла типа `Text`.

```
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);
   }
}
```



К значению, возвращаемому в результате выполнения метода `getData()`, рекомендуется применить метод `trim()`. Предположим, что разработчик XML-файла поместил открывающие и закрывающие дескрипторы в отдельных строках, как показано ниже.

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

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

Для извлечения последнего дочернего узла можно использовать метод `getLastChild()`, а для получения следующего узла того же уровня — метод `getNextSibling()`.

С помощью этих методов можно реализовать альтернативный способ обхода дочерних узлов.

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

Для перечисления атрибутов узла следует вызвать метод `getAttributes()`, который возвращает карту `NamedNodeMap` с объектами типа `Node`, описывающими атрибуты. Обход узлов карты `NamedNodeMap` можно выполнить точно так же, как и обход узлов списка `NodeList`. Затем для извлечения имен атрибутов и их значений нужно использовать методы `getNodeName()` и `getNodeValue()`.

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

Если известно имя атрибута, то его значение можно извлечь непосредственно:

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

Описанный здесь способ анализа дерева DOM демонстрирует программа, код которой приведен в листинге 2.1. Для чтения XML-файла следует выбрать пункт меню File⇨Open (Файл⇨Открыть) и указать в диалоговом окне требуемый файл. После этого объект `DocumentBuilder` выполнит анализ XML-данных и создаст объект `Document`. Затем этот объект будет показан в окне программы в виде древовидной структуры (рис. 2.3).

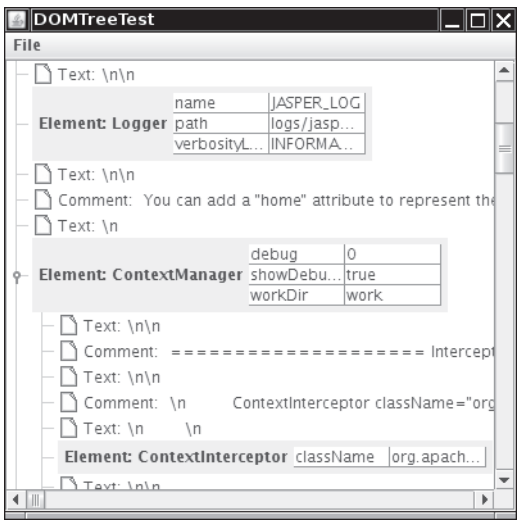


Рис. 2.3. Результат анализа XML-документа

В данной древовидной структуре видны дочерние узлы, которые окружены разделителями и комментариями. Для большей наглядности программа отображает все символы перевода строки и возврата каретки в виде `\n` и `\r`. (Иначе они были бы представлены в виде пустых квадратов; они используются средствами Swing для обозначения символов, которые невозможно воспроизвести.)

Здесь применяется рассматриваемый в главе 6 способ 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. Объект отображения ячеек дерева воспроизводит ряд компонентов дерева.

- Для элементов: 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 `\n` и `\r`.
- Для других типов узлов: имя класса, за которым следует результат выполнения метода `toString()`.

Листинг 2.1. Содержимое файла `DOMTreeTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.io.*;
4.  import javax.swing.*;
5.  import javax.swing.event.*;
6.  import javax.swing.table.*;
7.  import javax.swing.tree.*;
8.  import javax.xml.parsers.*;
9.  import org.w3c.dom.*;
10.
11. /**
12.    *Пример отображения XML-документа в виде древовидной структуры.
13.    *version 1.11 2007-06-24
14.    *author Cay Horstmann
15. */
16. public class DOMTreeTest
17. {
18.    public static void main(String[] args)
19.    {
20.       EventQueue.invokeLater(new Runnable()
21.         {
22.           public void run()
23.           {
24.             JFrame frame = new DOMTreeFrame();
25.             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26.             frame.setVisible(true);
27.           }
28.         });
29.    }
30. }
31. /**
32.  *Фрейм, содержащий древовидную структуру, посредством
33.   которой отображается содержимое XML-документа.
34. */
35. class DOMTreeFrame extends JFrame
36. {
37.    public DOMTreeFrame()
38.    {
39.       setTitle("DOMTreeTest");
40.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
41.
42.       JMenu fileMenu = new JMenu("File");
43.       JMenuItem openItem = new JMenuItem("Open");
44.       openItem.addActionListener(new ActionListener()
```

```
45.         {
46.            public void actionPerformed(ActionEvent event)
47.            {
48.               openFile();
49.            }
50.         });
51.      fileMenu.add(openItem);
52.
53.      JMenuItem exitItem = new JMenuItem("Exit");
54.      exitItem.addActionListener(new ActionListener()
55.         {
56.            public void actionPerformed(ActionEvent event)
57.            {
58.               System.exit(0);
59.            }
60.         });
61.      fileMenu.add(exitItem);
62.
63.      JMenuBar menuBar = new JMenuBar();
64.      menuBar.add(fileMenu);
65.      setJMenuBar(menuBar);
66.   }
67.   /**
68.    * Открытие файла и загрузка документа.
69.    */
70.   public void openFile()
71.   {
72.      JFileChooser chooser = new JFileChooser();
73.      chooser.setCurrentDirectory(new File("."));
74.
75.      chooser.setFileFilter(new javax.swing.filechooser.FileFilter()
76.         {
77.            public boolean accept(File f)
78.            {
79.               return f.isDirectory() || f.getName().toLowerCase().endsWith(".xml");
80.            }
81.
82.            public String getDescription()
83.            {
84.               return "XML files";
85.            }
86.         });
87.      int r = chooser.showOpenDialog(this);
88.      if (r != JFileChooser.APPROVE_OPTION) return;
89.      final File file = chooser.getSelectedFile();
90.
91.      new SwingWorker<Document, Void>()
92.         {
93.            protected Document doInBackground() throws Exception
94.            {
95.               if (builder == null)
96.                  {
97.                    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
98.                    builder = factory.newDocumentBuilder();
99.                  }
100.              return builder.parse(file);
101.           }
```

```
102.        protected void done()
103.        {
104.           try
105.           {
106.              Document doc = get();
107.              JTree tree = new JTree(new DOMTreeModel(doc));
108.              tree.setCellRenderer(new DOMTreeCellRenderer());
109.
110.              setContentPane(new JScrollPane(tree));
111.              validate();
112.           }
113.           catch (Exception e)
114.           {
115.              JOptionPane.showMessageDialog(DOMTreeFrame.this, e);
116.           }
117.        }
118.     }.execute();
119.  }
120.
121.  private DocumentBuilder builder;
122.  private static final int DEFAULT_WIDTH = 400;
123.  private static final int DEFAULT_HEIGHT = 400;
124. }
125.
126. /**
127.  * Данная модель дерева описывает структуру XML-документа.
128.  */
129. class DOMTreeModel implements TreeModel
130. {
131.    /**
132.     * Конструктор модели дерева для документа.
133.     * @param doc Документ
134.     */
135.    public DOMTreeModel(Document doc)
136.    {
137.       this.doc = doc;
138.    }
139.
140.    public Object getRoot()
141.    {
142.       return doc.getDocumentElement();
143.    }
144.
145.    public int getChildCount(Object parent)
146.    {
147.       Node node = (Node) parent;
148.       NodeList list = node.getChildNodes();
149.       return list.getLength();
150.    }
151.
152.    public Object getChild(Object parent, int index)
153.    {
154.       Node node = (Node) parent;
155.       NodeList list = node.getChildNodes();
156.       return list.item(index);
157.    }
158.
```

```
159.    public int getIndexOfChild(Object parent, Object child)
160.    {
161.       Node node = (Node) parent;
162.       NodeList list = node.getChildNodes();
163.       for (int i = 0; i < list.getLength(); i++)
164.          if (getChild(node, i) == child) return i;
165.       return -1;
166.    }
167.
168.    public boolean isLeaf(Object node)
169.    {
170.       return getChildCount(node) == 0;
171.    }
172.
173.    public void valueForPathChanged(TreePath path, Object newValue)
174.    {
175.    }
176.
177.    public void addTreeModelListener(TreeModelListener l)
178.    {
179.    }
180.
181.    public void removeTreeModelListener(TreeModelListener l)
182.    {
183.    }
184.
185.    private Document doc;
186. }
187.
188. /**
189.  * Данный класс извлекает узел XML.
190.  */
191. class DOMTreeCellRenderer extends DefaultTreeCellRenderer
192. {
193.    public Component getTreeCellRendererComponent(JTree tree,
194.       Object value, boolean selected,
195.       boolean expanded, boolean leaf, int row, boolean hasFocus)
196.    {
197.       Node node = (Node) value;
198.       if (node instanceof Element) return elementPanel((Element) node);
199.
200.       super.getTreeCellRendererComponent(tree, value, selected, expanded,
201.          leaf, row, hasFocus);
202.       if (node instanceof CharacterData) setText(characterString((CharacterData) node));
203.       else setText(node.getClass() + ": " + node.toString());
204.       return this;
205.    }
206.
207.    public static JPanel elementPanel(Element e)
208.    {
209.       JPanel panel = new JPanel();
210.       panel.add(new JLabel("Element: " + e.getTagName()));
211.       final NamedNodeMap map = e.getAttributes();
212.       panel.add(new JTable(new AbstractTableModel()
213.          {
214.             public int getRowCount()
215.             {
```

```
216.             return map.getLength();
217.          }
218.
219.          public int getColumnCount()
220.          {
221.             return 2;
222.          }
223.
224.          public Object getValueAt(int r, int c)
225.          {
226.             return c == 0 ? map.item(r).getNodeName() : map.item(r).getNodeValue();
227.          }
228.       }));
229.    return panel;
230. }
231.
232. public static String characterString(CharacterData node)
233. {
234.    StringBuilder builder = new StringBuilder(node.getData());
235.    for (int i = 0; i < builder.length(); i++)
236.    {
237.       if (builder.charAt(i) == '\r')
238.       {
239.          builder.replace(i, i + 1, "\\r");
240.          i++;
241.       }
242.       else if (builder.charAt(i) == '\n')
243.       {
244.          builder.replace(i, i + 1, "\\n");
245.          i++;
246.       }
247.       else if (builder.charAt(i) == '\t')
248.       {
249.          builder.replace(i, i + 1, "\\t");
250.          i++;
251.       }
252.    }
253.    if (node instanceof CDATASection) builder.insert(0, "CDATASection: ");
254.    else if (node instanceof Text) builder.insert(0, "Text: ");
255.    else if (node instanceof Comment) builder.insert(0, "Comment: ");
256.
257.    return builder.toString();
258. }
259. }
```

API javax.xml.parsers.DocumentBuilderFactory 1.4

- `static DocumentBuilderFactory newInstance()`
  Возвращает экземпляр класса `DocumentBuilderFactory`.
- `DocumentBuilder newDocumentBuilder()`
  Возвращает экземпляр класса `DocumentBuilder`.

API javax.xml.parsers.DocumentBuilder 1.4

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

  Выполняет разбор XML-документа, полученного из указанного файла, URL или входного потока. Возвращает результат разбора.

API org.w3c.dom.Document 1.4

- `Element getDocumentElement()`

  Возвращает корневой элемент документа.

API org.w3c.dom.Element 1.4

- `String getTagName()`

  Возвращает имя элемента.

- `String getAttribute(String name)`

  Возвращает значение атрибута с заданным именем либо пустую строку, если такого атрибута нет.

API org.w3c.dom.Node 1.4

- `NodeList getChildNodes()`

  Возвращает список, который содержит все узлы, дочерние по отношению к данному.

- `Node getFirstChild()`
- `Node getLastChild()`

  Возвращает первый или последний узел из числа дочерних по отношению к данному. Если данный узел не имеет дочерних узлов, возвращается значение `null`.

- `Node getNextSibling()`
- `Node getPreviousSibling()`

  Возвращает предыдущий узел того же уровня. Если для данного узла нет равноправных узлов, возвращается значение `null`.

- `Node getParentNode()`

  Возвращает узел, родительский по отношению к данному, либо значение `null`, если данный узел является узлом документа.

- `NamedNodeMap getAttributes()`

  Возвращает карту, которая содержит узлы типа `Attr` с описаниями всех атрибутов данного узла.

- `String getNodeName()`

  Возвращает имя данного узла. Если узел имеет тип `Attr`, то возвращается имя атрибута.

- `String getNodeValue()`

  Возвращает значение данного узла. Если узел имеет тип `Attr`, то возвращается значение атрибута.

API org.w3c.dom.CharacterData 1.4

- `String getData()`
  Возвращает строку, которая хранится в данном узле.

API 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`.

API org.w3c.dom.NamedNodeMap 1.4

- `int getLength()`
  Возвращает количество узлов в данной карте.
- `Node item(int index)`
  Возвращает узел с заданным индексом. Значение индекса может быть от `0` до `getLength()-1`.

Проверка XML-документов

В предыдущем разделе описан способ обхода древовидной структуры документа DOM. Однако если напрямую следовать этому подходу, то немало усилий придется приложить для организации проверки на наличие ошибок. При этом придется не только организовать поиск и удаление лишних разделителей между элементами, но также проверить, содержит ли документ нужные узлы. Рассмотрим приведенный ниже элемент.

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

Сначала будет получен первый дочерний узел. И сразу неожиданность. Это текстовый узел, содержащий разделитель "`\n  `". Пропускаем текстовые узлы и находим первый элемент. Необходимо проверить, имеет ли этот дескриптор имя `name`. Затем потребуется выяснить, имеет ли он дочерний узел типа `Text`. После этого перемещаемся к следующему дочернему узлу (который не является разделителем) и выполняем такую же проверку. Но что делать, если автор XML-документа изменит порядок расположения дочерних узлов или добавит новый дочерний элемент? С одной стороны, для проверки всех возможных ошибок придется создать очень громоздкий код, а с другой — исключить такую проверку было бы слишком опрометчиво.

К счастью, одной из положительных характеристик XML-анализатора является его способность автоматически проверять корректность структуры документа. В таком случае анализ XML-документа значительно упрощается. Например, если мы знаем, 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 нужно создать DTD либо XML-схему. И DTD, и схема содержат правила, которые регламентируют структуру документа. Они задают допустимые дочерние узлы элементов и атрибутов каждого элемента. Например, DTD-определение может содержать следующее правило:

```
<!ELEMENT font (name,size)>
```

Это правило отражает тот факт, что элемент `font` должен иметь два дочерних узла: `name` и `size`. С помощью XML-схемы то же 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>
```

XML-схема позволяет формулировать сложные правила, которые нельзя выразить средствами DTD. Например, с помощью схемы можно указать, что элемент `size` должен содержать целочисленное значение. В отличие от DTD, правила, соответствующие схеме, выражаются на языке XML, что упрощает их обработку.

XML-схема была изначально разработана для замены DTD. Однако на момент написания данной главы, средства DTD еще широко применялись. Язык схемы сложный, и еще не все анализаторы могут его поддерживать. На самом деле, некоторых пользователей настолько пугает сложность языка схемы, что они предпочитают применять альтернативные средства проверки корректности документов. В качестве примера такого средства можно привести язык Relax NG (`http://www.relaxng.org`).

В следующем разделе подробно обсуждается DTD, а затем вкратце рассматриваются основные характеристики XML-схемы. Затем вам будет предложен пример приложения, из которого станет ясно, что проверка корректности документа существенно упрощает XML-программирование.

Определения типов документов

Существует несколько способов связывания DTD с XML-документом. DTD можно включить в начало документ так, как показано ниже.

```
<?xml version="1.0" ?>
<!DOCTYPE configuration [
  <!ELEMENT configuration ...>
  другие правила
  ...
]>
<configuration>
  ...
</configuration>
```

Как видите, эти правила помещаются в квадратные скобки внутри объявления `DOCTYPE`. Тип документа должен соответствовать имени корневого элемента, т.е. `configuration` в данном примере. Размещение DTD внутри документа нельзя назвать наилучшим решением, так как объем определения может быть очень большим. Поэтому имеет смысл хранить DTD в отдельном файле. Для связывания таких DTD с документами можно использовать приведенные ниже объявления `SYSTEM`, посредством которых указываются имена внешних файлов.

```
<!DOCTYPE configuration SYSTEM "config.dtd">
<!DOCTYPE configuration SYSTEM "http://myserver.com/config.dtd">
```



Если для указания внешнего файла, содержащего DTD, используется относительный URL (например, "config.dtd"), то вместо InputStream анализатору нужно передать объект File или URL. Если же необходимо обеспечить разбор данных из входного потока, анализатору потребуется предоставить определитель примитивов.

Наконец, можно использовать механизм "хорошо известных" ("well known") DTD-определений, который унаследован из языка SGML:

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

Если XML-процессор знает способ обнаружения DTD с помощью идентификатора PUBLIC, то использовать URL не обязательно.



Если вы используете DOM-анализатор и хотите организовать поддержку идентификатора PUBLIC, следует вызвать метод setEntityResolver() класса DocumentBuilder. С помощью этого метода инсталлируется экземпляр класса, который реализует интерфейс EntityResolver. В данном интерфейсе определен единственный метод resolveEntity. Ниже показана типичная реализация этого метода.

```
class MyEntityResolver implements EntityResolver
{
    public InputSource resolveEntity(String publicID,
        String systemID)
    {
        if (publicID.equals(известный идентификатор))
            return new InputSource(DTD-данные)
        else
            return null; // использовать поведение по умолчанию
    }
}
```

Источник входных данных можно создать на основе класса InputStream, Reader или String.

Рассмотрим теперь различные правила, которые могут задаваться посредством DTD.

Правило ELEMENT указывает дочерние узлы данного элемента в виде регулярного выражения, которое состоит из перечисленных в табл. 2.1 компонентов.

Таблица 2.1. Правила для содержимого документа

Правило	Назначение
E*	0 или более элементов E
E+	1 или более элементов E
E?	0 или 1 элемент E
E_1|E_2|...|E_N	Один из элементов E_1, E_2, ..., E_N
E_1,E_2,...,E_N	Элемент E_1, за которым следуют E_2, ..., E_N
#PCDATA	Текст
(#PCDATA|E_1|E_2|...|E_N)*	0 или более текстовых элементов и элементы E_1, E_2, ..., E_N, расположенные в произвольном порядке (смешанное содержание)
ANY	Любой дочерний узел
EMPTY	Дочерние узлы отсутствуют

Рассмотрим несколько простых примеров. Следующее правило указывает на то, что элемент menu может содержать 0 или более элементов item:

```
<!ELEMENT menu (item)*>
```

Согласно приведенным ниже правилам, шрифт описывается именем, за которым следует размер. И имя, и размер являются текстовыми элементами.

```
<!ELEMENT font (name, size)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT size (#PCDATA)>
```

Аббревиатурой PCDATA обозначаются проанализированные символьные данные. Данные называются проанализированными, поскольку анализатор обрабатывает текстовую строку и ищет символ <, который обозначает начало нового дескриптора, или символ &, который обозначает начало примитива. Спецификация элемента может содержать регулярные выражения, в том числе вложенные. Например, ниже приведено правило, которое описывает структуру этой главы.

```
<!ELEMENT chapter (intro, (heading, (para|image|table|note)+)+)>
```

Каждая глава chapter начинается с введения intro, за которым располагается один или несколько разделов, состоящих из заголовка heading, одного или нескольких абзацев para, рисунков image, таблиц table или примечаний note.

Однако не всегда правила позволяют обеспечить достаточную гибкость. Довольно часто элементы содержат текст. Тогда возможны только два разных случая. Во-первых, в составе элемента может быть допустимо наличие только текста:

```
<!ELEMENT name (#PCDATA)>
```

Во-вторых, элемент может содержать *любое сочетание текста и дескрипторов, расположенных в произвольном порядке*:

```
<!ELEMENT para (#PCDATA|em|strong|code)*>
```

Если правило содержит #PCDATA, другие типы правил недопустимы. Например, приведенное ниже выражение некорректно.

```
<!ELEMENT captionedImage (image, #PCDATA)>
```

Такие правила следует переписать заново, либо вводя отдельный элемент caption, содержащий подпись, либо допуская любое сочетание дескрипторов image и текста.

Это ограничение упрощает работу XML-анализатора при разборе смешанного содержимого (текста и дескрипторов). Но в этом случае содержимое становится неконтролируемым, поэтому рекомендуется проектировать такие DTD-определения, которые содержали бы либо только элементы, либо только текст.



Утверждение о том, что в DTD-правилах можно указывать произвольные регулярные выражения, не совсем верно. 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, получив x, анализатор не сможет сразу выяснить, к какому из двух элементов он относится. Это выражение следует переписать в следующем виде: (x,(y|z)). Однако некоторые определения невозможно изменить, например ((x,y)*x?). Анализатор Sun не выдает никаких предупреждений при обнаружении подобных DTD-определений. Он просто применяет первый возможный вариант, что может привести к некорректной интерпретации корректных входных данных.

На практике определения, подобные приведенным выше, встречаются крайне редко, поскольку большинство DTD чрезвычайно просты.

Для описания допустимых атрибутов элементов используется приведенный ниже синтаксис.

```
<!ATTLIST элемент атрибут тип поведение_по_умолчанию>
```

В табл. 2.2 приведены допустимые типы атрибутов, а в табл. 2.3 — синтаксис параметра *поведение_по_умолчанию*.

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 несколько лексем, соответствующих имени
ID	Уникальный идентификатор
IDREF, IDREFS	Одна или несколько ссылок на уникальный идентификатор
ENTITY, ENTITIES	Один или несколько непроанализированных примитивов

Таблица 2.3. Поведение по умолчанию

Поведение	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, либо должен быть равен *A*; в любом случае анализатор возвращает значение *A*

Ниже показаны две типичных спецификации атрибутов.

```
<!ATTLIST font style (plain|bold|italic|bold-italic) "plain">
<!ATTLIST size unit CDATA #IMPLIED>
```

В первой спецификации для элемента `font` указан атрибут стиля `style`, который может иметь четыре допустимых значения; по умолчанию используется значение `plain`. Во второй спецификации для элемента `size` указан атрибут стиля `unit`, который может содержать любую последовательность символов.



Для описания данных рекомендуется применять элементы, а не атрибуты. В соответствии с этой рекомендацией стиль шрифта должен содержаться в отдельном элементе, например, в виде `<font><style>plain</style>...</font>`. Однако атрибуты обладают несомненным преимуществом при использовании перечислений, потому что анализатор может проверять допустимость тех или иных переменных. Например, если стиль шрифта является атрибутом, то анализатор проверяет наличие найденного стиля среди четырех указанных допустимых значений и использует значение по умолчанию, если допустимого найти не удалось.

Обработка атрибута типа `CDATA` несколько отличается от обработки атрибута `#PCDATA` и практически никак не связана с разделами `<![CDATA[...]]>`. Значение атрибута сначала *нормализуется*, т.е. анализатор обрабатывает ссылки на символы и примитивы (такие как, например, `é` или `<`) и заменяет разделители пробелами.

Тип NMTOKEN (лексема имени) аналогичен типу CDATA, но в нем не допускается использование большинства символов, отличных от букв и цифр, а также разделителей. Кроме того, удаляются все начальные и конечные разделители. Тип NMTOKENS представляет собой список лексем имен, между которыми включены разделители.

Тип атрибута ID означает лексему имени, уникальную для данного документа. Уникальность проверяется анализатором. Атрибут этого типа будет применяться в следующем примере. Тип IDREF означает ссылку на идентификатор, уже существующий в данном документе, наличие которого также проверяется анализатором. Атрибут типа IDREFS содержит список ссылок на идентификаторы.

Атрибут типа ENTITY указывает на "непроанализированный внешний примитив". Этот тип унаследован от языка SGML и редко используется на практике. В спецификации языка XML, расположенной по адресу http://www.xml.com/axml/axml.html, приводится пример использования такого типа атрибута.

DTD-определение может также содержать определения *примитивов*, или аббревиатур, которые заменяются в процессе разбора. Хороший пример использования примитивов можно найти в описании пользовательского интерфейса Web-браузера Mozilla/Netscape 6. Данное описание отформатировано в соответствии с требованиями XML и содержит определения примитивов, подобные приведенному ниже.

```
<!ENTITY back.label "Back">
```

Далее в тексте документа могут встретиться ссылки на эти примитивы.

```
<menuitem label="&back.label;"/>
```

В таком случае анализатор заменяет ссылку на примитив строкой замены. Для интернационализации приложения потребуется лишь изменить определение примитива. Примитивы также могут использоваться в других случаях. Более подробную информацию по этой теме можно найти в спецификации языка XML.

Итак, вы имеете основные сведения, позволяющие составить представление о DTD-определении. Рассмотрим теперь способы настройки анализатора для использования всех преимуществ DTD. Сначала нужно сообщить фабрике конструкторов документов о необходимости проверки:

```
factory.setValidating(true);
```

Все конструкторы, созданные этой фабрикой, проверяют соответствие входных данных DTD. Наиболее полезным результатом такой проверки является игнорирование всех разделителей в составе элемента. Рассмотрим фрагмент XML-кода:

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

Анализатор, не выполняющий проверку, вернет все разделители между элементами font, name и size. Он поступит так потому, что ему ничего не известно о том, какое из следующих правил описывает дочерние узлы элемента font:

```
(name,size)
(#PCDATA,name,size)*
ANY
```

Если в DTD указано правило (name,size), то анализатору наверняка известно, что разделитель между этими элементами не является текстом. Конструктор документа перестанет учитывать разделители, если использовать следующий метод:

```
factory.setIgnoringElementContentWhitespace(true);
```

Теперь можно быть *вполне уверенным* в том, что узел `font` имеет два дочерних узла, и не использовать приведенный ниже громоздкий код.

```
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 него для доступа к первому и второму дочерним узлам можно использовать следующие строки:

```
Element nameElement = (Element)children.item(0);
Element sizeElement = (Element)children.item(1);
```

Именно в этих случаях DTD-определения особенно полезны. Как видите, при использовании DTD уже не нужно включать в программу сложные фрагменты кода, выполняющие проверку, потому что анализатор проделал эту работу во время получения документа.



Многие программисты, начиная работу с XML-документами, игнорируют 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 приложение должно каким-то образом отреагировать на это, например, зарегистрировать ошибку, сообщить о ней пользователю или сгенерировать исключение для прекращения разбора. Поэтому при использовании средств проверки содержимого XML-документа следует также установить обработчик исключений, для чего создать объект, реализующий интерфейс `ErrorHandler`. В этом интерфейсе объявлены три метода:

```
void warning(SAXParseException exception)
void error(SAXParseException exception)
void fatalError(SAXParseException exception)
```

Для установки обработчика исключений используется метод `setErrorHandler()` класса `DocumentBuilder`:

```
builder.setErrorHandler(handler);
```

API javax.xml.parsers.DocumentBuilder 1.4

- `void setEntityResolver(EntityResolver resolver)`
  Устанавливает определитель примитивов, которые упоминаются в анализируемых XML-документах.
- `void setErrorHandler(ErrorHandler handler)`
  Устанавливает обработчик исключений для выдачи предупреждений и сообщений об ошибках, возникающих при разборе XML-документов.

API org.xml.sax.EntityResolver 1.4

- `public InputSource resolveEntity(String publicID, String systemID)`

  Возвращает источник входной информации, который содержит данные, определяемые заданными идентификаторами, или значение `null`, указывающее на то, что этот определитель не знает, как обработать данное имя. Параметр `publicID` может иметь значение `null`, если нет общедоступных идентификаторов.

API org.xml.sax.InputSource 1.4

- `InputSource(InputStream in)`
- `InputSource(Reader in)`
- `InputSource(String systemID)`

  Создает источник входных данных на основании указанного потока объекта `Reader` или системного идентификатора (обычно это относительный или абсолютный URL).

API org.xml.sax.ErrorHandler 1.4

- `void fatalError(SAXParseException exception)`
- `void error(SAXParseException exception)`
- `void warning(SAXParseException exception)`

  Данные методы нужно переопределить для создания собственного обработчика неустранимых ошибок, устранимых ошибок или предупреждений.

API org.xml.sax.SAXParseException 1.4

- `int getLineNumber()`
- `int getColumnNumber()`

  Возвращает номер строки или столбца в конце данных из входного потока, при обработке которых возникло исключение.

API javax.xml.parsers.DocumentBuilderFactory 1.4

- `boolean isValidating()`
- `void setValidating(boolean value)`

  Возвращает или устанавливает признак проверки для фабрики. Значение `true` указывает на то, что созданные фабрикой анализаторы будут выполнять проверку входных данных.

- `boolean isIgnoringElementContentWhitespace()`
- `void setIgnoringElementContentWhitespace(boolean value)`

  Возвращает или устанавливает значение, определяющее, следует ли игнорировать разделители между элементами. Значение `true` указывает на то, что созданные фабрикой анализаторы будут игнорировать разделители в случае, если для элемента не задано смешанное содержимое.

XML-схема

Поскольку XML-схема гораздо сложнее DTD, рассмотрим лишь основные ее характеристики. Дополнительную информацию можно найти в учебном руководстве, расположенном по адресу: `http://www.w3.org/TR/xmlschema-0`.

Для того чтобы включить в документ ссылку на файл схемы, нужно указать в корневом элементе соответствующие атрибуты, например:

```
<?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 содержится информация о том, что при проверке документа должен использоваться файл схемы `config.xsd`. Если в документе применяются пространства имен, синтаксис несколько усложняется. Подробно об этом можно прочитать в учебном руководстве, URL которого приведен выше. (Префикс `xsi` представляет собой псевдоним пространства имен — более подробно об этом см. раздел “Использование пространств имен”.)

Схема определяет тип каждого элемента. Это может быть простой тип (строка с ограничениями на форматирование) или сложный тип. Некоторые простые типы предопределены в XML-схеме. Примеры таких типов приведены ниже.

```
xsd:string
xsd:int
xsd:boolean
```



Мы используем префикс `xsd:`, обозначающий пространство имен XML Schema Definition. Некоторые авторы применяют для этой же цели префикс `xs:`.

При необходимости вы можете определить собственные простые типы. Пример подобного определения приведен ниже:

```
<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, следует указать его тип:

```
<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, результат проверки следующих элементов будет положительным:

```
<size>10</size>
<style>PLAIN</style>
```

В отличие от них, элементы, приведенные ниже, будут отвергнуты 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>
```

Здесь FontType — это последовательность элементов name, size и style. В данном определении использован атрибут ref, посредством которого мы ссылаемся на определения, находящиеся в схеме. Возможны также вложенные определения, подобные показанному ниже:

```
<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, что для элемента style использовано *безымянное*, или *анонимное определение типа* (anonymous type definition).

Элемент xsd:sequence является аналогом конкатенации в DTD. Элемент xsd:choice можно использовать вместо операции |.

```
<xsd:complexType name="contactinfo">
  <xsd:choice>
    <xsd:element ref="email"/>
    <xsd:element ref="phone"/>
  </xsd:choice>
</xsd:complexType>
```

Данная конструкция эквивалентна DTD-типу email|phone.

Для повторяющихся элементов можно использовать атрибуты minoccurs и maxoccurs. Например, аналогом DTD-типа item* является следующее выражение:

```
<xsd:element name="item" type=". . ." minoccurs="0" maxoccurs="unbounded">
```

Для определения атрибутов необходимо включить в определение complexType элементы xsd:attribute:

```
<xsd:element name="size">
  <xsd:complexType>
    . . .
    <xsd:attribute name="unit" type="xsd:string" use="optional"
      default="cm"/>
  </xsd:complexType>
</xsd:element>
```

Аналогом приведенной выше конструкции является следующее DTD-выражение:

```
<!ATTLIST size unit CDATA #IMPLIED "cm">
```

Определения элемента и типа помещаются в состав элемента `xsd:schema`:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   . . .
</xsd:schema>
```

Разбор XML-файла, формат которого определяется схемой, выполняется практически так же, как и разбор файла, для которого используется DTD. Отличия заключаются в следующем.

- Даже если вы не используете пространства имен в XML-файлах, следует включить средства их поддержки.

  ```
  factory.setNamespaceAware(true);
  ```

- Необходимо подготовить фабрику для обработки схем. 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);
  ```

- Анализатор *не отвергает разделители в составе элемента*. Это серьезное упущение, относительно которого существует разногласие о том, является ли это ошибкой, или нет. Способ решения этой проблемы представлен в листинге 2.4.

Практический пример XML-документа

В данном разделе рассматривается пример использования XML-документа при решении реальной задачи. В главе 9 первого тома уже отмечалось, что класс `GridBagLayout` лучше всего подходит для размещения Swing-компонентов. Однако многие опасаются его применять не столько из-за 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 описываются способы использования XML при работе с диспетчером `GridBagLayout` и анализа файлов с подобным описанием.

Элементы, скомпонованные с помощью диспетчера `GridBagLayout`, размещаются в строках и столбцах. Порядок их расположения очень похож на HTML-таблицу, поэтому мы опишем структуру в виде последовательности строк, каждая из которых содержит ячейки.

```
<gridbag>
   <row>
      <cell>...</cell>
      <cell>...</cell>
      ...
   </row>
   <row>
      <cell>...</cell>
      <cell>...</cell>
      ...
   </row>
   ...
</gridbag>
```

В файле gridbag.dtd укажем следующие правила:

```
<!ELEMENT gridbag (row)*>
<!ELEMENT row (cell)*>
```

Некоторые ячейки могут распространяться на несколько строк и столбцов. В диспетчере компоновки GridBagLayout это достигается путем установки значений gridwidth и gridheight, превышающих 1. Мы используем атрибуты с теми же именами:

```
<cell gridwidth="2" gridheight="2">
```

Аналогично введем атрибуты, определяющие другие ограничения: fill, anchor, gridx, gridy, weightx, weighty, ipadx и ipady. (Ограничение insets не используется в данном примере, так как его нельзя представить с помощью простого типа и организовать поддержку будет сложно.)

```
<cell fill="HORIZONTAL" anchor="NORTH">
```

Для большинства атрибутов используются те же значения по умолчанию, которые предлагаются в конструкторе 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 при указании их вручную часто могут возникать ошибки:

```
<!ATTLIST cell gridx CDATA #IMPLIED>
<!ATTLIST cell gridy CDATA #IMPLIED>
```

Как видите, эти значения задавать не обязательно; в случае их отсутствия программа поступит следующим образом: в столбце 0 для gridx по умолчанию будет использоваться значение 0, а во всех других столбцах — предыдущее значение gridx плюс предыдущее значение gridwidth. По умолчанию значение gridy равно номеру строки. Поэтому в большинстве случаев значения gridx и gridy задавать не придется.



Программисты, имеющие опыт работы с GridBagLayout, скажут, что для автоматического определения значений gridx и gridy можно было бы применять параметры RELATIVE и REMAINDER. Мы попытались это сделать, но расположение элементов, аналогичное диалоговому окну выбора шрифта, которое было показано на рис. 2.4, получить не удалось. После внимательного изучения исходного кода GridBagLayout стало ясно, что данный класс попросту не способен выполнить тонкую работу по определению абсолютных позиций.

В процессе работы программа анализирует атрибуты и задает ограничения для диспетчера компоновки GridBagLayout. Например, для определения gridwidth используется следующее выражение:

```
constraints.gridwidth = Integer.parseInt(e.getAttribute("gridwidth"));
```

Программе не нужно беспокоиться о пропущенных атрибутах, поскольку анализатор автоматически предоставит значение по умолчанию, если оно не будет обнаружено в документе.

Для проверки наличия атрибутов gridx и gridy следует вызвать метод getAttribute() и проверить, возвращает ли он пустую строку:

```
String value = e.getAttribute("gridy");
  if (value.length() == 0) // Используется по умолчанию.
    constraints.gridy = r;
  else
    constraints.gridx = Integer.parseInt(value);
```

Мы выяснили, что удобнее будет предоставить пользователю возможность указывать произвольные объекты внутри ячеек. Это позволило бы использовать даже такой тип, как, например, граница, который не является компонентом. Нужно лишь, чтобы объекты принадлежали классу, отвечающему требованиям компонента JavaBeans – он должен содержать конструктор по умолчанию и иметь свойства, формирующиеся парами getter/setter (получатель/установщик). (Компоненты JavaBeans рассматриваются в главе 8.) Компонент JavaBeans определяется именем класса и свойствами:

```
<!ELEMENT bean (class, property*)>
<!ELEMENT class (#PCDATA)>
```

Свойство содержит имя name и значение value.

```
<!ELEMENT property (name, value)>
<!ELEMENT name (#PCDATA)>
```

Значением value может быть целое число, логическое значение, строка или компонент:

```
<!ELEMENT value (int|string|boolean|bean)>
<!ELEMENT int (#PCDATA)>
<!ELEMENT string (#PCDATA)>
<!ELEMENT boolean (#PCDATA)>
```

Ниже в качестве примера приведен компонент JLabel со свойством text, которое имеет значение "Face: ".

```
<bean>
  <class>javax.swing.JLabel</class>
  <property>
    <name>text</name>
    <value><string>Face: </string></value>
  </property>
</bean>
```

На первый взгляд может показаться странным, что строка окружена дескрипторами <string>. Почему бы не использовать параметр #PCDATA для строк, а дескрипторы применять только для других типов? Дело в том, что в таком случае придется использовать смешанное содержание и менее строгое правило для элемента value:

```
<!ELEMENT value ((#PCDATA|int|boolean|bean)*>
```

Значение свойства задается в программе с помощью класса BeanInfo, в котором перечисляются описания свойств компонента. Мы ищем свойство с соответствующим именем, затем вызываем его set-метод, задавая требуемое значение в качестве параметра.

После считывания описания пользовательского интерфейса программа получает достаточное количество информации для создания компонентов пользовательского интерфейса и их размещения. Но, конечно, такой интерфейс не будет активным, поскольку в нем нет обработчиков событий. Для создания обработчиков необходимо уметь находить компоненты. Поэтому для каждого компонента bean поддерживается необязательный атрибут ID:

```
<!ATTLIST bean id ID #IMPLIED>
```

Например, описание раскрывающегося списка с указанием идентификатора выглядит следующим образом:

```
<bean id="face">
  <class>javax.swing.JComboBox</class>
</bean>
```

Напомним, что анализатор проверяет уникальность идентификаторов.
Программист может присоединять обработчики событий, как показано ниже.

```
gridbag = new GridBagPane("fontdialog.xml");
setContentPane(gridbag);
JComboBox face = (JComboBox)gridbag.get("face");
face.addListener(listener);
```



В этом примере мы используем XML только для описания компоновки компонента, а программисты могут сами добавлять в Java-код обработчики событий. Вы же можете пойти еще дальше, включив код в XML-описание. Лучше всего для этой цели применить язык написания сценариев, такой как JavaScript. Если вы все-таки захотите сделать это, попробуйте воспользоваться интерпретатором Rhino (`http://www.mozilla.org/rhino`).

Программа, код которой показан в листинге 2.2, демонстрирует способ использования класса `GridBagPane` для выполнения всей трудоемкой и рутинной работы по установке характеристик диспетчера компоновки `GridBagLayout`. Расположение компонентов определяется в XML-файле, содержимое которого приведено в листинге 2.3. Внешний вид контейнера, с которым связан диспетчер `GridBagLayout`, показан на рис. 2.4. Программа инициализирует только раскрывающиеся списки (которые слишком сложны для того, чтобы применять к ним механизм установки свойств компонентов JavaBeans, поддерживаемый классом `GridBagPane`) и присоединяет обработчики событий. Класс `GridBagPane`, код которого приведен в листинге 2.4, анализирует XML-файл, создает компоненты и располагает их в окне программы. В листинге 2.5 представлено содержимое файла с DTD.



Рис. 2.4. Диалоговое окно выбора шрифта, для которого расположение компонентов определяется в XML-файле

Программа также поддерживает XML-схему. Для работы со схемой ее нужно запустить следующим образом:

```
java GridBagTest fontdialog-schema.xml
```

Код схемы приведен в листинге 2.6.

В данной программе демонстрируется типичный пример использования XML. Этот язык позволяет представлять достаточно сложные взаимосвязи между объектами. XML-анализатор берет на себя рутинную работу по проверке корректности документа и установке значений по умолчанию.

Листинг 2.2. Содержимое файла `GridBagTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import javax.swing.*;
4.
5.  /**
6.   *Пример использования XML-файла для описания расположения элементов.
7.   * @version 1.01 2007-06-25
8.   * @author Cay Hortsmann
9.   */
10. public class GridBagTest
11. {
12.   public static void main(final String[] args)
13.   {
14.      EventQueue.invokeLater(new Runnable()
15.         {
16.            public void run()
17.            {
18.               String filename = args.length == 0 ? "fontdialog.xml" : args[0];
19.               JFrame frame = new FontFrame(filename);
20.               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.               frame.setVisible(true);
22.            }
23.         });
24.   }
25. }
26.
27. /**
28.  * Фрейм, представляющий диалоговое окно выбора шрифта,
29.  * для расположения элементов которого используется XML-файл.
30.  * @param filename Файл, содержащий компоненты интерфейса
31.  */
32. class FontFrame extends JFrame
33. {
34.   public FontFrame(String filename)
35.   {
36.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
37.      setTitle("GridBagTest");
38.
39.      gridbag = new GridBagPane(filename);
40.      add(gridbag);
41.
42.      face = (JComboBox) gridbag.get("face");
43.      size = (JComboBox) gridbag.get("size");
44.      bold = (JCheckBox) gridbag.get("bold");
45.      italic = (JCheckBox) gridbag.get("italic");
```

```
46.       face.setModel(new DefaultComboBoxModel(new Object[] { "Serif",
47.          "SansSerif", "Monospaced", "Dialog", "DialogInput" }));
48.
49.       size.setModel(new DefaultComboBoxModel(new Object[] { "8", "10", "12",
50.          "15", "18", "24", "36", "48" }));
51.
52.       ActionListener listener = new
53.          ActionListener()
54.          {
55.             public void actionPerformed(ActionEvent event)
56.                { setSample(); }
57.          };
58.
59.       face.addActionListener(listener);
60.       size.addActionListener(listener);
61.       bold.addActionListener(listener);
62.       italic.addActionListener(listener);
63.
64.       setSample();
65.    }
66.
67.    /**
68.     * Формирование образца текста для демонстрации
69.     * выбранного шрифта.
70.     */
71.    public void setSample()
72.    {
73.       String fontFace = (String) face.getSelectedItem();
74.       int fontSize =
75.          Integer.parseInt((String) size.getSelectedItem());
76.       JTextArea sample = (JTextArea) gridbag.get("sample");
77.       int fontStyle =
78.          (bold.isSelected() ? Font.BOLD : 0)
79.          + (italic.isSelected() ? Font.ITALIC : 0);
80.
81.       sample.setFont(new Font(fontFace, fontStyle, fontSize));
82.       sample.repaint();
83.    }
84.
85.    private GridBagPane gridbag;
86.    private JComboBox face;
87.    private JComboBox size;
88.    private JCheckBox bold;
89.    private JCheckBox italic;
90.    private static final int DEFAULT_WIDTH = 400;
91.    private static final int DEFAULT_HEIGHT = 400;
92. }
```

Листинг 2.3. Содержимое файла `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"
20.         weighty="100">
21.         <bean id="sample">
22.           <class>javax.swing.JTextArea</class>
23.           <property>
24.             <name>text</name>
25.             <value>
26.             <string>The quick brown fox jumps over the lazy dog
27.             </string></value>
28.           </property>
29.           <property>
30.             <name>editable</name>
31.             <value><boolean>false</boolean></value>
32.           </property>
33.           <property>
34.             <name>lineWrap</name>
35.             <value><boolean>true</boolean></value>
36.           </property>
37.           <property>
38.             <name>border</name>
39.             <value>
40.               <bean>
41.                <class>javax.swing.border.EtchedBorder</class>
42.               </bean>
43.             </value>
44.           </property>
45.         </bean>
46.       </cell>
47.     </row>
48.     <row>
49.       <cell anchor="EAST">
50.         <bean>
51.           <class>javax.swing.JLabel</class>
52.           <property>
53.             <name>text</name>
54.             <value><string>Size: </string></value>
55.           </property>
56.         </bean>
57.       </cell>
58.       <cell fill="HORIZONTAL" weightx="100">
59.         <bean id="size">
60.           <class>javax.swing.JComboBox</class>
61.         </bean>
62.       </cell>
63.     </row>
64.     <row>
```

```
65.       <cell gridwidth="2" weighty="100">
66.         <bean id="bold">
67.           <class>javax.swing.JCheckBox</class>
68.           <property>
69.             <name>text</name>
70.             <value><string>Bold</string></value>
71.           </property>
72.         </bean>
73.       </cell>
74.     </row>
75.     <row>
76.       <cell gridwidth="2" weighty="100">
77.         <bean id="italic">
78.           <class>javax.swing.JCheckBox</class>
79.           <property>
80.             <name>text</name>
81.             <value><string>Italic</string></value>
82.           </property>
83.         </bean>
84.       </cell>
85.     </row>
86.  </gridbag>
```

Листинг 2.4. Содержимое файла `GridBagPane.java`

```
1.   import java.awt.*;
2.   import java.beans.*;
3.   import java.io.*;
4.   import java.lang.reflect.*;
5.   import javax.swing.*;
6.   import javax.xml.parsers.*;
7.   import org.w3c.dom.*;
8.
9.   /**
10.   * Данная панель использует XML-файл для описания
11.   * компонентов и их расположения в сетке, создаваемой
12.   * диспетчером компоновки GridBagLayout.
13.   * @version 1.10 2004-09-04
14.   * @author Cay Horstmann
15.   */
16.  public class GridBagPane extends JPanel
17.  {
18.    /**
19.     * Конструктор класса GridBagPane.
20.     * @param filename Имя XML-файла, описывающего компоненты
21.     * и их расположение
22.     */
23.    public GridBagPane(String filename)
24.    {
25.      setLayout(new GridBagLayout());
26.      constraints = new GridBagConstraints();
27.
28.      try
29.      {
30.        DocumentBuilderFactory factory =
31.          DocumentBuilderFactory.newInstance();
32.        factory.setValidating(true);
```

```
33.         if (filename.contains("-schema"))
34.         {
35.           factory.setNamespaceAware(true);
36.           final String JAXP_SCHEMA_LANGUAGE =
37.             "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
38.           final String W3C_XML_SCHEMA =
39.             "http://www.w3.org/2001/XMLSchema";
40.           factory.setAttribute(JAXP_SCHEMA_LANGUAGE,
41.             W3C_XML_SCHEMA);
42.         }
43.
44.         factory.setIgnoringElementContentWhitespace(true);
45.
46.         DocumentBuilder builder = factory.newDocumentBuilder();
47.         Document doc = builder.parse(new File(filename));
48.
49.         if (filename.contains("-schema"))
50.         {
51.           int count =
52.             removeElementContentWhitespace(doc.getDocumentElement());
53.           System.out.println(count +
54.             " whitespace nodes removed.");
55.         }
56.
57.         parseGridbag(doc.getDocumentElement());
58.       }
59.       catch (Exception e)
60.       {
61.         e.printStackTrace();
62.       }
63.     }
64.
65.     /**
66.      * Удаление всех (определяется эвристическим способом) разделителей
67.      * в содержимом элемента.
68.      * @param e Корневой элемент
69.      * @return Число удаленных узлов, соответствующих разделителям.
70.      */
71.     private int removeElementContentWhitespace(Element e)
72.     {
73.       NodeList children = e.getChildNodes();
74.       int count = 0;
75.       boolean allTextChildrenAreWhiteSpace = true;
76.       int elements = 0;
77.       for (int i = 0; i < children.getLength() && allTextChildrenAreWhiteSpace; i++)
78.       {
79.         Node child = children.item(i);
80.         if (child instanceof Text && ((Text) child).getData().trim().length() > 0)
81.           allTextChildrenAreWhiteSpace = false;
82.         else if (child instanceof Element)
83.         {
84.           elements++;
85.           count += removeElementContentWhitespace((Element) child);
86.         }
87.       }
```

```
88.       if (elements > 0 && allTextChildrenAreWhiteSpace)
89.                               // эвристика для содержимого элемента
90.       {
91.          for (int i = children.getLength() - 1; i >= 0; i--)
92.          {
93.             Node child = children.item(i);
94.             if (child instanceof Text)
95.             {
96.                e.removeChild(child);
97.                count++;
98.             }
99.          }
100.      }
101.      return count;
102.   }
103.
104.   /**
105.    * Получение компонента с заданным именем.
106.    * @param name Имя компонента
107.    * @return Компонент или значение null, если в составе панели
108.    * отсутствует компонент с указанным именем
109.    */
110.   public Component get(String name)
111.   {
112.      Component[] components = getComponents();
113.      for (int i = 0; i < components.length; i++)
114.      {
115.         if (components[i].getName().equals(name)) return components[i];
116.      }
117.      return null;
118.   }
119.
120.   /**
121.    * Разбор элемента gridbag.
122.    * @param e Элемент gridbag
123.    */
124.   private void parseGridbag(Element e)
125.   {
126.      NodeList rows = e.getChildNodes();
127.      for (int i = 0; i < rows.getLength(); i++)
128.      {
129.         Element row = (Element) rows.item(i);
130.         NodeList cells = row.getChildNodes();
131.         for (int j = 0; j < cells.getLength(); j++)
132.         {
133.            Element cell = (Element) cells.item(j);
134.            parseCell(cell, i, j);
135.         }
136.      }
137.   }
138.
139.   /**
140.    * Разбор элемента cell.
141.    * @param e Элемент cell.
142.    * @param r Строка, которой принадлежит элемент cell.
143.    * @param c Столбец, которому принадлежит элемент cell.
144.    */
```

```
145.   private void parseCell(Element e, int r, int c)
146.   {
147.      // Получение атрибутов.
148.
149.      String value = e.getAttribute("gridx");
150.      if (value.length() == 0) // использование умолчаний
151.      {
152.         if (c == 0) constraints.gridx = 0;
153.         else constraints.gridx += constraints.gridwidth;
154.      }
155.      else constraints.gridx = Integer.parseInt(value);
156.
157.      value = e.getAttribute("gridy");
158.      if (value.length() == 0) // использование умолчаний
159.      constraints.gridy = r;
160.      else constraints.gridy = Integer.parseInt(value);
161.
162.      constraints.gridwidth = Integer.parseInt(e.getAttribute("gridwidth"));
163.      constraints.gridheight = Integer.parseInt(e.getAttribute("gridheight"));
164.      constraints.weightx = Integer.parseInt(e.getAttribute("weightx"));
165.      constraints.weighty = Integer.parseInt(e.getAttribute("weighty"));
166.      constraints.ipadx = Integer.parseInt(e.getAttribute("ipadx"));
167.      constraints.ipady = Integer.parseInt(e.getAttribute("ipady"));
168.
169.      // Использование отражения для получения целочисленных
170.      // значений статических полей.
171.      Class<GridBagConstraints> cl = GridBagConstraints.class;
172.
173.      try
174.      {
175.         String name = e.getAttribute("fill");
176.         Field f = cl.getField(name);
177.         constraints.fill = f.getInt(cl);
178.
179.         name = e.getAttribute("anchor");
180.         f = cl.getField(name);
181.         constraints.anchor = f.getInt(cl);
182.      }
183.      catch (Exception ex) // При отражении могут возникать различные исключения.
184.      {
185.         ex.printStackTrace();
186.      }
187.      Component comp = (Component) parseBean((Element) e.getFirstChild());
188.      add(comp, constraints);
189.   }
190.
191.   /**
192.    * Разбор элемента bean.
193.    * @param e Элемент bean
194.    */
195.   private Object parseBean(Element e)
196.   {
197.      try
198.      {
199.         NodeList children = e.getChildNodes();
200.         Element classElement = (Element) children.item(0);
201.         String className = ((Text) classElement.getFirstChild()).getData();
```

```
202.
203.        Class<?> cl = Class.forName(className);
204.
205.        Object obj = cl.newInstance();
206.
207.        if (obj instanceof Component) ((Component) obj).setName(e.getAttribute("id"));
208.
209.        for (int i = 1; i < children.getLength(); i++)
210.        {
211.          Node propertyElement = children.item(i);
212.          Element nameElement = (Element) propertyElement.getFirstChild();
213.          String propertyName = ((Text) nameElement.getFirstChild()).getData();
214.
215.          Element valueElement = (Element) propertyElement.getLastChild();
216.          Object value = parseValue(valueElement);
217.          BeanInfo beanInfo = Introspector.getBeanInfo(cl);
218.          PropertyDescriptor[] descriptors = beanInfo.getPropertyDescriptors();
219.          boolean done = false;
220.          for (int j = 0; !done && j < descriptors.length; j++)
221.          {
222.            if (descriptors[j].getName().equals(propertyName))
223.            {
224.              descriptors[j].getWriteMethod().invoke(obj, value);
225.              done = true;
226.            }
227.          }
228.        }
229.        return obj;
230.      }
231.      catch (Exception ex) // При отражении могут возникать различные исключения.
232.      {
233.        ex.printStackTrace();
234.        return null;
235.      }
236.    }
237.
238.    /**
239.     * Разбор элемента value.
240.     * @param e Элемент value
241.     */
242.    private Object parseValue(Element e)
243.    {
244.      Element child = (Element) e.getFirstChild();
245.      if (child.getTagName().equals("bean")) return parseBean(child);
246.      String text = ((Text) child.getFirstChild()).getData();
247.      if (child.getTagName().equals("int")) return new Integer(text);
248.      else if (child.getTagName().equals("boolean")) return new Boolean(text);
249.      else if (child.getTagName().equals("string")) return text;
250.      else return null;
251.    }
252.
253.    private GridBagConstraints constraints;
254. }
```

Листинг 2.5. Содержимое файла `gridbag.dtd`

```
<!ELEMENT gridbag (row)*>
<!ELEMENT row (cell)*>
<!ELEMENT cell (bean)>
<!ATTLIST cell gridx CDATA #IMPLIED>
<!ATTLIST cell gridy CDATA #IMPLIED>
<!ATTLIST cell gridwidth CDATA "1">
<!ATTLIST cell gridheight CDATA "1">
<!ATTLIST cell weightx CDATA "0">
<!ATTLIST cell weighty CDATA "0">
<!ATTLIST cell fill (NONE|BOTH|HORIZONTAL|VERTICAL) "NONE">
<!ATTLIST cell anchor (CENTER|NORTH|NORTHEAST|EAST|
   SOUTHEAST|SOUTH|SOUTHWEST|WEST|NORTHWEST) "CENTER">
<!ATTLIST cell ipadx CDATA "0">
<!ATTLIST cell ipady CDATA "0">

<!ELEMENT bean (class, property*)>
<!ATTLIST bean id ID #IMPLIED>

<!ELEMENT class (#PCDATA)>
<!ELEMENT property (name, value)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT value (int|string|boolean|bean)>
<!ELEMENT int (#PCDATA)>
<!ELEMENT string (#PCDATA)>
<!ELEMENT boolean (#PCDATA)>
```

Листинг 2.6. Содержимое файла `gridbag.xsd`

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="gridbag" type="GridBagType"/>

  <xsd:element name="bean" type="BeanType"/>

  <xsd:complexType name="GridBagType">
    <xsd:sequence>
      <xsd:element name="row" type="RowType" minOccurs="0"
         maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="RowType">
    <xsd:sequence>
      <xsd:element name="cell" type="CellType" minOccurs="0"
         maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="CellType">
    <xsd:sequence>
      <xsd:element ref="bean"/>
    </xsd:sequence>
    <xsd:attribute name="gridx" type="xsd:int" use="optional"/>
    <xsd:attribute name="gridy" type="xsd:int" use="optional"/>
    <xsd:attribute name="gridwidth" type="xsd:int" use="optional" default="1" />
    <xsd:attribute name="gridheight" type="xsd:int" use="optional" default="1" />
```

```
28.       <xsd:attribute name="weightx" type="xsd:int" use="optional" default="0" />
29.       <xsd:attribute name="weighty" type="xsd:int" use="optional" default="0" />
30.       <xsd:attribute name="fill" use="optional" default="NONE">
31.         <xsd:simpleType>
32.           <xsd:restriction base="xsd:string">
33.             <xsd:enumeration value="NONE" />
34.             <xsd:enumeration value="BOTH" />
35.             <xsd:enumeration value="HORIZONTAL" />
36.             <xsd:enumeration value="VERTICAL" />
37.           </xsd:restriction>
38.         </xsd:simpleType>
39.       </xsd:attribute>
40.       <xsd:attribute name="anchor" use="optional"
41.          default="CENTER">
42.         <xsd:simpleType>
43.           <xsd:restriction base="xsd:string">
44.             <xsd:enumeration value="CENTER" />
45.             <xsd:enumeration value="NORTH" />
46.             <xsd:enumeration value="NORTHEAST" />
47.             <xsd:enumeration value="EAST" />
48.             <xsd:enumeration value="SOUTHEAST" />
49.             <xsd:enumeration value="SOUTH" />
50.             <xsd:enumeration value="SOUTHWEST" />
51.             <xsd:enumeration value="WEST" />
52.             <xsd:enumeration value="NORTHWEST" />
53.           </xsd:restriction>
54.         </xsd:simpleType>
55.       </xsd:attribute>
56.       <xsd:attribute name="ipady" type="xsd:int" use="optional" default="0" />
57.       <xsd:attribute name="ipadx" type="xsd:int" use="optional" default="0" />
58.     </xsd:complexType>
59.
60.     <xsd:complexType name="BeanType">
61.       <xsd:sequence>
62.         <xsd:element name="class" type="xsd:string"/>
63.         <xsd:element name="property" type="PropertyType"
64.            minOccurs="0" maxOccurs="unbounded"/>
65.       </xsd:sequence>
66.       <xsd:attribute name="id" type="xsd:ID" use="optional" />
67.     </xsd:complexType>
68.
69.     <xsd:complexType name="PropertyType">
70.       <xsd:sequence>
71.         <xsd:element name="name" type="xsd:string"/>
72.         <xsd:element name="value" type="ValueType"/>
73.       </xsd:sequence>
74.     </xsd:complexType>
75.
76.     <xsd:complexType name="ValueType">
77.       <xsd:choice>
78.         <xsd:element ref="bean"/>
79.         <xsd:element name="int" type="xsd:int"/>
80.         <xsd:element name="string" type="xsd:string"/>
81.         <xsd:element name="boolean" type="xsd:boolean"/>
82.       </xsd:choice>
83.     </xsd:complexType>
84.  </xsd:schema>
```

Поиск информации с помощью XPath

Если вам необходимо найти информацию в XML-документе, то обязательно придется организовать обход древовидной структуры DOM. Язык XPath упрощает доступ к узлам дерева. Предположим, например, что в вашем распоряжении находится XML-документ, приведенный ниже.

```
<configuration>
   . . .
   <database>
      <username>dbuser</username>
      <password>secret</password>
      . . .
   </database>
</configuration>
```

Вам необходимо определить имя пользователя, обрабатывая выражение XPath:

```
/configuration/database/username
```

При наличии такого выражения поиск становится несколько проще, чем путем непосредственного анализа дерева DOM.

1. Получение узла документа.
2. Перечисление его дочерних узлов.
3. Обнаружение элемента `database`.
4. Получение первого дочернего элемента, `username`.
5. Получение первого дочернего узла, который имеет тип `Text`.
6. Получение данных.

Язык XPath позволяет описывать набор узлов в XML-документе. Например, следующее выражение описывает набор элементов `row`, которые являются дочерними по отношению к корневому элементу:

```
/gridbag/row
```

Для выбора конкретного элемента используется операция `[]`. Следующее выражение определяет первый дочерний элемент (отчет индексов начинается с единицы):

```
/gridbag/row[1]
```

Для получения значений атрибутов применяется операция `@`. Приведенное ниже выражение XPath описывает все узлы, соответствующие атрибутам `anchor` элементов `cell`. Элементы `cell` должны входить в состав элементов `row`, которые являются дочерними по отношению к корневому элементу `gridbag`.

```
/gridbag/row/cell/@anchor
```

Существует ряд функций XPath, упрощающих работу с документом. Например, показанное ниже выражение позволяет определить число элементов `row`, дочерних по отношению к корневому элементу `gridbag`:

```
count(/gridbag/row)
```

Примеры выражений XPath, в том числе довольно сложных, можно найти в спецификации `http://www.w3c.org/TR/xpath`. Существует также очень удачно составленное руководство, расположенное по адресу `http://www.zvon.org/xxl/XPathTutorial/General/examples.html`.

В состав Java SE 5.0 включен API для обработки выражений XPath. В первую очередь надо создать объект `XPath`, используя для этого класс `XPathFactory`:

```
XPathFactory xpfactory = XPathFactory.newInstance();
path = xpfactory.newXPath();
```

После этого можно вызывать метод `evaluate()`, обрабатывающий выражения XPath:

```
String username = path.evaluate("/configuration/database/username", doc);
```

Один объект `XPath` можно применять для обработки нескольких выражений.

Данный вариант метода `evaluate()` возвращает результат в виде строки. Это удобно для получения текста, каковым является содержимое узла `username` в предыдущем примере. Если выражение `XPath` описывает набор узлов, то для его обработки можно использовать следующий вызов:

```
NodeList nodes = (NodeList) path.evaluate("/gridbag/row", doc,
   XPathConstants.NODESET);
```

Если результатом является один узел, то в качестве третьего параметра метода `evaluate()` нужно указать значение `XPathConstants.NODE`:

```
Node node = (Node) path.evaluate("/gridbag/row[1]", doc,
   XPathConstants.NODE);
```

Если результатом является число, применяется константа `XPathConstants.NUMBER`:

```
int count = ((Number) path.evaluate("count(/gridbag/row)", doc,
    XPathConstants.NUMBER)).intValue();
```

Начинать поиск не обязательно с корневого узла документа. В качестве исходной точки может выступать любой узел, и даже список узлов. Например, получив узел в результате обработки приведенного выше выражения, можно вызвать метод `evaluate()` следующим образом:

```
result = path.evaluate(expression, node);
```

Программа, код которой приведен в листинге 2.7, иллюстрирует процесс обработки выражений XPath. Загрузите XML-файл и введите выражение. Выберите тип выражения и щелкните на кнопке `Evaluate`. Результат обработки выражения отобразится в нижней части окна (рис. 2.5).

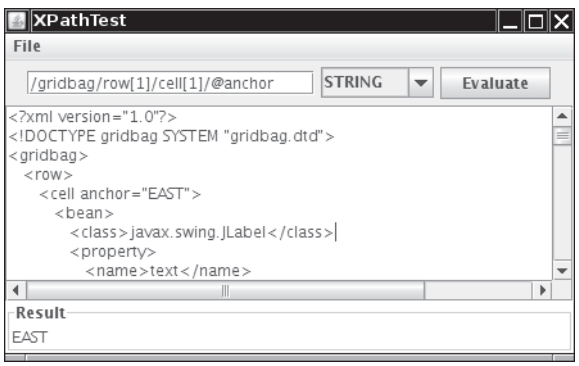


Рис. 2.5. Обработка выражений XPath

Листинг 2.7. Содержимое файла XPathTest.java

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.io.*;
4.  import javax.swing.*;
5.  import javax.swing.border.*;
6.  import javax.xml.namespace.*;
7.  import javax.xml.parsers.*;
8.  import javax.xml.xpath.*;
9.  import org.w3c.dom.*;
10. import org.xml.sax.*;
11.
12. /**
13.  * Программа для обработки выражений XPath.
14.  * @version 1.01 2007-06-25
15.  * @author Cay Horstmann
16.  */
17. public class XPathTest
18. {
19.    public static void main(String[] args)
20.    {
21.       EventQueue.invokeLater(new Runnable)
22.          {
23.             public void run()
24.             {
25.                JFrame frame = new XPathFrame();
26.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27.                frame.setVisible(true);
28.             }
29.          });
30.    }
31. }
32.
33. /**
34.  * В данном фрейме отображается XML-документ, панель для ввода
35.  * выражений XPath и поле редактирования для отображения
36.  * результатов.
37.  */
38. class XPathFrame extends JFrame
39. {
40.    public XPathFrame()
41.    {
42.       setTitle("XPathTest");
43.
44.       JMenu fileMenu = new JMenu("File");
45.       JMenuItem openItem = new JMenuItem("Open");
46.       openItem.addActionListener(new
47.          ActionListener()
48.          {
49.             public void actionPerformed(ActionEvent event)
50.                {
51.                   openFile(); }
52.                }
53.        });
54.       fileMenu.add(openItem);
55.       JMenuItem exitItem = new JMenuItem("Exit");
```

```
56.       exitItem.addActionListener(new ActionListener()
57.         {
58.           public void actionPerformed(ActionEvent event)
59.           {
60.             System.exit(0);
61.           }
62.         });
63.       fileMenu.add(exitItem);
64.
65.       JMenuBar menuBar = new JMenuBar();
66.       menuBar.add(fileMenu);
67.       setJMenuBar(menuBar);
68.
69.       ActionListener listener = new
70.         ActionListener()
71.         {
72.           public void actionPerformed(ActionEvent event)
73.           {
74.             evaluate();
75.           }
76.         };
77.       expression = new JTextField(20);
78.       expression.addActionListener(listener);
79.       JButton evaluateButton = new JButton("Evaluate");
80.       evaluateButton.addActionListener(listener);
81.
82.       typeCombo = new JComboBox(new Object[]
83.         { "STRING", "NODE", "NODESET", "NUMBER", "BOOLEAN" });
84.       typeCombo.setSelectedItem("STRING");
85.
86.       JPanel panel = new JPanel();
87.       panel.add(expression);
88.       panel.add(typeCombo);
89.       panel.add(evaluateButton);
90.       docText = new JTextArea(10, 40);
91.       result = new JTextField();
92.       result.setBorder(new TitledBorder("Result"));
93.
94.       add(panel, BorderLayout.NORTH);
95.       add(new JScrollPane(docText), BorderLayout.CENTER);
96.       add(result, BorderLayout.SOUTH);
97.
98.       try
99.       {
100.        DocumentBuilderFactory factory =
101.          DocumentBuilderFactory.newInstance();
102.        builder = factory.newDocumentBuilder();
103.      }
104.      catch (ParserConfigurationException e)
105.      {
106.        JOptionPane.showMessageDialog(this, e);
107.      }
108.
109.      XPathFactory xpfactory = XPathFactory.newInstance();
110.      path = xpfactory.newXPath();
111.      pack();
112.    }
```

```
113.   /**
114.    * Открытие файла и загрузка документов.
115.    */
116.   public void openFile()
117.   {
118.      JFileChooser chooser = new JFileChooser();
119.      chooser.setCurrentDirectory(new File("."));
120.
121.      chooser.setFileFilter(new
122.         javax.swing.filechooser.FileFilter()
123.         {
124.            public boolean accept(File f)
125.            {
126.               return f.isDirectory() ||
127.                     f.getName().toLowerCase().endsWith(".xml");
128.            }
129.
130.            public String getDescription()
131.            {
132.               return "XML files";
133.            }
134.         });
135.      int r = chooser.showOpenDialog(this);
136.      if (r != JFileChooser.APPROVE_OPTION) return;
137.      File f = chooser.getSelectedFile();
138.      try
139.      {
140.         byte[] bytes = new byte[(int) f.length()];
141.         new FileInputStream(f).read(bytes);
142.         docText.setText(new String(bytes));
143.         doc = builder.parse(f);
144.      }
145.      catch (IOException e)
146.      {
147.         JOptionPane.showMessageDialog(this, e);
148.      }
149.      catch (SAXException e)
150.      {
151.         JOptionPane.showMessageDialog(this, e);
152.      }
153.   }
154.
155.   public void evaluate()
156.   {
157.      try
158.      {
159.         String typeName = (String) typeCombo.getSelectedItem();
160.         QName returnType = (QName)
161.            XPathConstants.class.getField(typeName).get(null);
162.         Object evalResult = path.evaluate(expression.getText(),
163.            doc, returnType);
164.         if (typeName.equals("NODESET"))
165.         {
166.            NodeList list = (NodeList) evalResult;
167.            StringBuilder builder = new StringBuilder();
168.            builder.append("{");
169.            for (int i = 0; i < list.getLength(); i++)
```

```
170.           {
171.             if (i > 0) builder.append(", ");
172.             builder.append("" + list.item(i));
173.           }
174.           builder.append("}");
175.           result.setText("" + builder);
176.         }
177.         else result.setText("" + evalResult);
178.       }
179.       catch (XPathExpressionException e)
180.       {
181.         result.setText("" + e);
182.       }
183.       catch (Exception e) // Исключение, связанное с отражением
184.       {
185.         e.printStackTrace();
186.       }
187.    }
188.
189.    private DocumentBuilder builder;
190.    private Document doc;
191.    private XPath path;
192.    private JTextField expression;
193.    private JTextField result;
194.    private JTextArea docText;
195.    private JComboBox typeCombo;
196. }
```

API javax.xml.xpath.XPathFactory 5.0

- `static XPathFactory newInstance()`

  Возвращает экземпляр класса `XPathFactory`, используемый для создания объектов `XPath`.

- `XPath newXpath()`

  Создает объект `XPath`, который можно использовать для обработки выражений XPath.

API javax.xml.xpath.XPath 5.0

- `String evaluate(String expression, Object startingPoint)`

  Обрабатывает выражение, начиная поиск с заданной точки. В качестве исходной точки может быть указан узел или список узлов. Если результатом обработки выражения является узел или набор узлов, то возвращаемая строка содержит данные из всех дочерних текстовых узлов.

- `Object evaluate(String expression, Object startingPoint, QName resultType)`

  Обрабатывает выражение, начиная поиск с заданной точки. В качестве исходной точки может быть указан узел или список узлов. В качестве параметра `resultType` задается одна из следующих констант, определенных в классе `XPathConstants`: `STRING`, `NODE`, `NODESET`, `NUMBER` или `BOOLEAN`. Возвращаемое значение имеет тип `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 находятся в разных пакетах. В XML для элементов и атрибутов используются *пространства имен* (namespace).

Пространство имен идентифицируется с помощью URI, например, так, как показано ниже:

```
http://www.w3c.org/2001/XMLSchema
uuid:1c759aed-b748-475c-ab68-10679700c4f2
urn:com:books-r-us
```

Чаще всего для этой цели используется формат URL. Учтите, что в данном случае URL лишь выполняет роль идентификатора. Приведенные ниже URL обозначают *разные* пространства имен, хотя Web-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, может не указывать на конкретный документ; XML-анализатор и не пытается найти что-либо по этому адресу. Однако принято располагать по URL, идентифицирующему пространство имен, документ с его описанием. Например, по адресу `http://www.w3c.org/2001/XMLSchema` находится документ с описанием XML-схем.

Почему для обозначения пространства имен применяются URL? Очевидно, что в таком случае легче гарантировать их уникальность. Действительно, для реального URL уникальность имени хоста сети гарантируется структурой системы доменных имен, а уникальность остальной части URL должна обеспечить организация файловой системы. Именно из этих соображений для многих пакетов выбраны доменные имена с обратным порядком следования доменов.

Конечно, для длинных имен проще обеспечить уникальность, однако использовать их в программе не совсем удобно. В языке Java для этого предусмотрен механизм импортирования пакетов, в результате чего в исходном тексте программы в основном присутствуют лишь имена классов. В XML предусмотрены подобные средства. Рассмотрим приведенное ниже выражение:

```
<элемент xmlns="URI_пространства_имен">
   дочерние_узлы
</элемент>
```

В результате элемент и все его дочерние узлы становятся частью заданного пространства имен.

При необходимости дочерний узел может обеспечить себе отдельное пространство имен:

```
<элемент xmlns="URI_пространства_имен">
   <дочерний_узел xmlns="URI_пространства_имен_2">
      дочерние_узлы_следующего_уровня
   </дочерний_узел>
   дочерние_узлы
</элемент>
```

Здесь первый дочерний узел и дочерние узлы следующего уровня принадлежат второму пространству имен.

Это простое решение подходит только в том случае, если требуется одно пространство имен или если пространства имен вложены друг в друга. В других ситуациях предпочтительнее альтернативный механизм, который не имеет аналога в языке Java. Для пространства имен можно использовать *псевдоним* (alias), т.е. короткий идентификатор, действующий только в пределах документа.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="gridbag" type="GridBagType"/>
  . . .
</xsd:schema>
```

Атрибут `xmlns:`*`псевдоним`*`="`*`URI_пространства_имен`*`"` определяет пространство имен и псевдоним. В этом примере псевдонимом является строка `xsd`. Таким образом, `xsd:schema` фактически означает, что "`schema` находится в пространстве имен `http://www.w3.org/2001/XMLSchema`".



Пространство имен родительского элемента наследуется только дочерними элементами. Атрибуты без явного указания префикса не считаются частью пространства имен. Рассмотрим 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 `configuration` и `size` являются частью пространства имен, определяемого URI `http://www.horstmann.com/corejava`. Атрибут `si:unit` является частью пространства имен с URI `http://www.bipm.fr/enus/3_SI/si.html`. Однако атрибут `value` не принадлежит ни одному из этих пространств имен.

То, как анализатор распоряжается пространствами имен, поддается контролю. По умолчанию DOM-анализатор производства Sun не учитывает пространства имен.

Для включения режима управления пространствами имен нужно вызвать метод `setNamespaceAware()` класса `DocumentBuilderFactory`:

```
factory.setNamespaceAware(true);
```

После этого все созданные данной фабрикой конструкторы будут поддерживать пространства имен.

Каждый узел имеет три свойства.

- *Квалифицированное имя* (qualified name) с префиксом-псевдонимом, возвращаемое методами `getNodeName()`, `getTagName()` и т.д.
- URI пространства имен, возвращаемый методом `getNamespaceURI()`.
- *Локальное имя* (local name) без префикса псевдонима, возвращаемое методом `getLocalName()`.

Предположим, что анализатор получает следующий элемент:

```
<xsd:schema xmlns:xsl="http://www.w3.org/2001/XMLSchema">
```

В этом случае он сообщит о наличии таких компонентов:

- квалифицированное имя — `xsd:schema`;
- URI пространства имен — `http://www.w3.org/2001/XMLSchema`;
- локальное имя — `schema`.



Если режим управления пространствами имен отключен, то методы `getNamespaceURI()` и `getLocalName()` возвращают значение `null`.



`org.w3c.dom.Node 1.4`

- `String getLocalName()`
  Возвращает локальное имя (без префикса) или значение `null`, если режим управления пространствами имен отключен.
- `String getNamespaceURI()`
  Возвращает URI пространства имен или 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)`
  Возвращает или устанавливает значение свойства, определяющего режим управления пространствами имен. Если данное значение равно `true`, то в генерируемых фабрикой анализаторах будет включен режим управления пространствами имен.

Потоковые анализаторы

DOM-анализатор считывает XML-документ и представляет его как древовидную структуру. Для большинства приложений этот принцип работы вполне подходит. Однако он совершенно неэффективен при обработке большого документа в том случае, если алгоритм не предполагает просмотра всего дерева. Именно в таких ситуациях оправдано применение потоковых анализаторов.

В следующих разделах мы рассмотрим потоковые анализаторы, доступные в библиотеке Java: знакомый всем SAX-анализатор и более современный StAX-анализатор, который появился в Java SE 6. SAX-анализатор использует обратные вызовы событий, а StAX-анализатор предлагает механизм итераций событий анализа. Последний анализатор является более удобным в использовании.

Использование SAX-анализатора

SAX-анализатор генерирует события в ходе разбора XML-документа. В общем случае документ не хранится в памяти, лишь обработчики событий могут создать структуру для хранения полученных данных. Фактически в основе работы DOM-анализатора лежит тот же принцип, который используется в SAX-анализаторе: дерево создается в результате обработки событий, сгенерированных в процессе разбора.

Для использования SAX-анализатора необходимо создать обработчик событий, которые происходят во время разбора XML-документа. В интерфейсе `ContentHandler` определено несколько методов обратного вызова, к которым обращается анализатор.

- Методы `startElement()` и `endElement()` вызываются каждый раз при получении открывающего и закрывающего дескрипторов.
- Метод `characters()` вызывается при получении символьных данных.
- Методы `startDocument()` и `endDocument()` вызываются в начале и в конце документа.

Рассмотрим фрагмент XML-документа:

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

Во время анализа этого фрагмента SAX-анализатор генерирует обратные вызовы нескольких методов.

1. `startElement()` — для элемента `font`.
2. `startElement()` — для элемента `name`.
3. `characters()` — для строки `Helvetica`.
4. `endElement()` — для элемента `name`.
5. `startElement()` — для элемента `size` с атрибутом `units="pt"`.
6. `characters()` — для строки `36`.
7. `endElement()` — для элемента `size`.
8. `endElement()` — для элемента `font`.

Для выполнения требуемых действий во время вызовов этих методов их необходимо переопределить. Программа, код которой приведен в листинге 2.8, выводит информацию 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 путем перехода по ссылкам автоматически выявляют новые Web-страницы, подлежащие индексации.



К сожалению, большинство HTML-страниц совершенно не соответствует формату XML, и данная программа не способна обработать их. Как уже отмечалось выше, консорциум W3C рекомендует Web-дизайнерам использовать язык XHTML, т.е. диалект HTML, который поддерживается современными Web-браузерами и призван уменьшить разрыв между HTML и XML. Все Web-страницы W3C написаны на языке XHTML, и их можно применять для проверки данной тестовой программы. Например, для создания списка всех гипертекстовых ссылок на Web-странице `http://www.w3c.org/MarkUp` нужно использовать следующую команду:

```
java SAXTest http://www.w3c.org/MarkUp
```

Программа, рассматриваемая здесь в качестве примера, иллюстрирует использование SAX-анализатора. В ней полностью игнорируется контекст, в котором находится элемент `a`, а также не сохраняется древовидная структура документа.

Для получения SAX-анализатора используется приведенная ниже команда:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
```

Для обработки документа следует использовать такую команду:

```
parser.parse(source, handler);
```

Здесь `source` — это источник входных данных, который может быть файлом, URL или входным потоком; `handler` — подкласс класса `DefaultHandler`. В классе `DefaultHandler` определены методы, объявленные в четырех интерфейсах:

```
ContentHandler
DTDHandler
EntityResolver
ErrorHandler
```

Эти методы не выполняют никаких действий.

В программе определен обработчик событий, в котором для поиска элементов с именем `a` и атрибутом `href` переопределен метод `startElement()` интерфейса `ContentHahdler`:

```
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, которые описывают имя элемента. Параметр `qname` сообщает определенное имя в формате *`псевдоним`*:*`локальное_имя`*. Если включен режим использования пространства имен, то параметры `namespaceURI` и `lname` описывают пространство имен и локальное имя.

Как и при работе с DOM-анализатором, режим использования пространства имен изначально отключен. Он включается с помощью вызова метода `setNamespaceAware()` класса-фабрики `SAXParserFactory`:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setNamespaceAware(true);
SAXParser saxParser = factory.newSAXParser();
```

Итак, в листинге 2.8 содержится код простейшего сетевого робота. Далее в главе рассматривается еще один интересный пример применения SAX-анализатора. Простой способ превращения источника данных, несовместимого с XML, в XML-документ — это генерация SAX-событий, которые может обрабатывать анализатор. Более подробно эти вопросы рассматриваются в разделе, посвященном XSL-преобразованию.

Листинг 2.8. Содержимое файла `SAXTest.java`

```
1.  import java.io.*;
2.  import java.net.*;
3.  import javax.xml.parsers.*;
4.  import org.xml.sax.*;
5.  import org.xml.sax.helpers.*;
6.
7.  /**
8.   * Данная программа демонстрирует использование SAX-анализатора.
9.   * В результате работы программы выводится информация о
10.  * гипертекстовых ссылках, содержащихся в XHTML-документе.
11.  * Запуск программы: java SAXTest url
12.  * @version 1.00 2001-09-29
13.  * @author Cay Horstmann
14.  */
```

```
15. public class SAXTest
16. {
17.    public static void main(String[] args) throws Exception
18.    {
19.       String url;
20.       if (args.length == 0)
21.       {
22.          url = "http://www.w3c.org";
23.          System.out.println("Using " + url);
24.       }
25.       else url = args[0];
26.       DefaultHandler handler = new DefaultHandler()
27.          {
28.             public void startElement(String namespaceURI, String lname,
29.                 String qname, Attributes attrs)
30.             {
31.                if (lname.equalsIgnoreCase("a") && attrs != null)
32.                {
33.                   for (int i = 0; i < attrs.getLength(); i++)
34.                   {
35.                      String aname = attrs.getLocalName(i);
36.                      if (aname.equalsIgnoreCase("href"))
37.                         System.out.println(attrs.getValue(i));
38.                   }
39.                }
40.             }
41.          };
42.       SAXParserFactory factory = SAXParserFactory.newInstance();
43.       factory.setNamespaceAware(true);
44.       SAXParser saxParser = factory.newSAXParser();
45.       InputStream in = new URL(url).openStream();
46.       saxParser.parse(in, handler);
47.    }
48. }
```

API 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)`
  Возвращает или устанавливает значение свойства фабрики, определяющего режим управления пространством имен. Если данное свойство равно `true`, то для генерируемых фабрикой анализаторов включен режим управления пространствами имен.
- `boolean isValidating()`
- `void setValidating(boolean value)`
  Возвращает или устанавливает значение свойства, управляющего проверкой входных данных. Если это свойство равно `true`, то для генерируемых фабрикой анализаторов включен режим проверки входных данных.

API javax.xml.parsers.SAXParser 1.4

- `void parse(File f, DefaultHandler handler)`
- `void parse(String url, DefaultHandler handler)`
- `void parse(InputStream in, DefaultHandler handler)`

  Анализирует XML-код, полученный из файла по указанному URL или из входного потока, и оповещает заданный обработчик о возникающих событиях.

API org.xml.sax.ContentHandler 1.4

- `void startDocument()`
- `void endDocument()`

  Вызывается в начале и в конце XML-документа.

- `void startElement(String uri, String lname, String qname, Attributes attr)`
- `void endElement(String uri, String lname, String qname)`

  Вызывается в начале и в конце элемента.

  Параметры:

`uri`	URI пространства имен (если включен режим использования пространства имен).
`lname`	Локальное имя без префикса-псевдонима (если включен режим использования пространства имен).
`qname`	Имя элемента (если отключен режим использования пространства имен) или определенное имя с префиксом-псевдонимом (если анализатор сообщает определенные 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.
`length`	Длина полученной строки.

API org.xml.sax.Attributes 1.4

- `int getLength()`

  Возвращает количество атрибутов, которые содержатся в наборе.

- `String getLocalName(int index)`

  Возвращает локальное имя (без префикса-псевдонима) атрибута с заданным индексом или пустую строку, если для анализатора не включен режим использования пространства имен.

- `String getURI(int index)`

  Возвращает URI пространства имен для атрибута с заданным индексом или пустую строку, если элемент не является частью пространства имен или для анализатора не включен соответствующий режим.

- `String getQName(int index)`
  Возвращает определенное имя (с префиксом) атрибута, соответствующего заданному индексу, или пустую строку, если определенное имя не предоставлено анализатором.
- `String getValue(int index)`
- `String getValue(String qname)`
- `String getValue(String uri, String lname)`
  Возвращает значение атрибута, заданного посредством индекса, определенного имени или URI пространства имен и локального имени. Если такое значение отсутствует, возвращается `null`.

Использование StAX-анализатора

StAX-анализатор является "извлекающим анализатором" (pull parser). Вместо того чтобы устанавливать обработчик события, вы просто производите перебор событий посредством следующего цикла:

```
InputStream in = url.openStream();
XMLInputFactory factory = XMLInputFactory.newInstance();
XMLStreamReader parser = factory.createXMLStreamReader(in);
while (parser.hasNext())
{
   int event = parser.next();
   Вызвать методы анализатора для получения подробных сведений о событии
}
```

Например, в процессе производства анализа следующего фрагмента:

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

анализатор дает следующие 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, нужно вызвать соответствующие методы класса `XMLStreamReader`. Например,

```
String units = parser.getAttributeValue(null, "units");
```

получает атрибут `units` текущего элемента.

По умолчанию обработка пространств имен включена. Отключить ее можно посредством изменения фабрики:

```
XMLInputFactory factory = XMLInputFactory.newInstance();
factory.setProperty(XMLInputFactory.IS_NAMESPACE_AWARE, false);
```

В листинге 2.9 содержится код программы “Web-ползунка” (Web crawler), реализованной посредством StAX-анализатора. Как можно видеть, этот код гораздо проще, чем эквивалентный SAX-код, поскольку в данном случае вам не нужно заботиться об обработке событий.

Листинг 2.9. Содержимое файла `StAXTest.java`

```
1.  import java.io.*;
2.  import java.net.*;
3.  import javax.xml.stream.*;
4.
5.  /**
6.   * Данная программа демонстрирует пример использования StAX-анализатора.
7.   * Программа выводит все гиперссылки, имеющиеся на Web-странице XHTML.
8.   * Запуск программы: java StAXTest url
9.   * @author Cay Horstmann
10.  * @version 1.0 2007-06-23
11.
12.  */
13. public class StAXTest
14. {
15.    public static void main(String[] args) throws Exception
16.    {
17.       String urlString;
18.       if (args.length == 0)
19.       {
20.          urlString = "http://www.w3c.org";
21.          System.out.println("Using " + urlString);
22.       }
23.       else urlString = args[0];
24.
25.       URL url = new URL(urlString);
26.       InputStream in = url.openStream();
27.       XMLInputFactory factory = XMLInputFactory.newInstance();
28.       XMLStreamReader parser = factory.createXMLStreamReader(in);
29.
30.       while (parser.hasNext())
31.       {
32.          int event = parser.next();
33.          if (event == XMLStreamConstants.START_ELEMENT)
34.          {
35.             if (parser.getLocalName().equals("a"))
36.             {
37.                String href = parser.getAttributeValue(null, "href");
38.                if (href != null)
39.                   System.out.println(href);
40.             }
41.          }
42.       }
43.    }
44. }
```

API javax.xml.stream.XMLInputFactory

- `static XMLInputFactory newInstance()`

  Возвращает экземпляр класса `XMLInputFactory`.

- `void setProperty(String имя, Object значение)`

  Задает свойство для данной фабрики, или генерирует исключение `IllegalArgumentException`, если свойство не поддерживается, или ему не может быть присвоено данное значение. Реализация Java SE поддерживает следующие допустимые свойства `Boolean`:

`"javax.xml.stream.isValidating"`	Если это свойство имеет значение `false` (оно присваивается по умолчанию), документ не проверяется. Спецификация не требует этого свойства.
`"javax.xml.stream.isNamespaceAware"`	Если это свойство имеет значение `true` (оно присваивается по умолчанию), производится обработка пространств имен. Спецификация не требует этого свойства.
`"javax.xml.stream.isCoalescing"`	Если это свойство имеет значение `false` (оно присваивается по умолчанию), соседние символы не объединяются.
`"javax.xml.stream.isReplacingEntityReferences"`	Если это свойство имеет значение `true` (оно присваивается по умолчанию), ссылки на примитивы заменяются, и о них сообщается как о данных символов.
`"javax.xml.stream.isSupportingExternalEntities"`	Если это свойство имеет значение `true` (оно присваивается по умолчанию), производится разрешение внешних примитивов. Спецификация не определяет никаких значений по умолчанию для этого свойства.
`"javax.xml.stream.supportDTD"`	Если это свойство имеет значение `true` (оно присваивается по умолчанию), о DTD сообщается как о событиях.

- `XMLStreamReader createXMLStreamReader(InputStream in)`
- `XMLStreamReader createXMLStreamReader(InputStream in, String characterEncoding)`
- `XMLStreamReader createXMLStreamReader(Reader in)`
- `XMLStreamReader createXMLStreamReader(Source in)`

  Создает анализатор, который производит чтение из данного потока, читателя или источника AXP.

API javax.xml.stream.XMLStreamReader

- `boolean hasNext()`
  Возвращает значение `true`, если существует другое событие анализатора.
- `int next()`
  Задает состояние анализатора следующему событию анализатора 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()`
  Возвращает значение `true`, если текущее событие является начальным элементом, конечным элементом, данными символов или пробелом.
- `QName getName()`
- `String getLocalName()`
  Получает имя элемента в событии `START_ELEMENT` или `END_ELEMENT`.
- `String getText()`
  Возвращает символы события `CHARACTERS`, `COMMENT` или `CDATA`, заменяемое значение для `ENTITY_REFERENCE` или внутреннюю подборку `DTD`.
- `int getAttributeCount()`
- `QName getAttributeName(int индекс)`
- `String getAttributeLocalName(int индекс)`
- `String getAttributeValue(int индекс)`
  Получает подсчет количества атрибутов, а также имена и значения атрибутов, при условии, что текущим событием является `START_ELEMENT`.
- `String getAttributeValue(String namespaceURI, String имя)`
  Получает значение атрибута по данному имени, при условии, что текущим событием является `START_ELEMENT`. Если `namespaceURI` имеет значение `null`, пространство имен не проверяется.

Генерирование XML-документов

После изучения способов написания Java-программ, предназначенных для чтения XML-документов, рассмотрим программы, которые выполняют обратное действие, т.е. генерируют данные в формате XML. Конечно, XML-документ можно сформировать, включив в программу последовательность вызовов `print()` и выводя с их помощью элементы, атрибуты и текст. Но для создания такого громоздкого кода потребуется много времени и усилий. Кроме того, подобный код, как правило, изобилует ошибками, например, легко ошибиться при использовании специальных символов " или < в значениях атрибутов и в тексте.

Удобнее создать древовидную структуру DOM, представляющую документ, а затем записать содержимое этого дерева в файл. В первую очередь следует сгенерировать пустой документ с помощью метода `newDocument()` класса `DocumentBuilder`:

```
Document doc = builder.newDocument();
```

Затем для создания элементов документа нужно применить метод `createElement()` класса `Document`:

```
Element rootElement = doc.createElement(rootName);
Element childElement = doc.createElement(childName);
```

Далее создаем текстовые узлы с помощью метода `createTextNode()`:

```
Text textNode = doc.createTextNode(textContents);
```

Для включения в документ корневого элемента и для связывания дочерних узлов с родительскими используется приведенный ниже код:

```
doc.appendChild(rootElement);
rootElement.appendChild(childElement);
childElement.appendChild(textNode);
```

При формировании дерева DOM атрибуты элементов задаются с помощью метода `setAttribute()` класса `Element`:

```
rootElement.setAttribute(name, value);
```

Как ни странно, в текущей версии API DOM не предусмотрены средства записи дерева DOM в выходной поток. Для преодоления этого ограничения следует использовать API-интерфейс XSLT (Extensible Stylesheet Language Transformations — трансформации расширяемого языка стилей). Более подробно XSLT-преобразования описываются в следующем разделе, а пока предположим, что этот код каким-то "магическим" образом позволяет получить выходные данные в виде XML-документа. К документу применяется преобразование, в процессе которого не выполняется никаких действий и результат записывается в выходной поток. Для включения узла `DOCTYPE` в состав выходных данных нужно указать в качестве свойств идентификаторы `SYSTEM` и `PUBLIC`.

```
// Преобразование, не выполняющее никаких действий.
Transformer t = TransformerFactory.newInstance().newTransformer();
// Установка свойств для получения узла DOCTYPE.
t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, systemIdentifier);
t.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC, publicIdentifier);
// Применение отступа.
t.setOutputProperty(OutputKeys.INDENT, "yes");
t.setOutputProperty(OutputKeys.METHOD, "xml");
t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
// Применение преобразования, не выполняющего никаких действий,
// и запись результатов в файл.
t.transform(new DOMSource(doc), new StreamResult(new FileOutputStream(file)));
```



Полученный XML-документ не содержит разделителей (т.е. символов перевода строки, табуляции или пробелов). Для включения их в состав документа следует указать значение `yes` для свойства `OutputKeys.INDENT`.

В листинге 2.10 приведен код программы, которая позволяет получить XML-документ. Эта программа рисует "картину в стиле модерн", т.е. произвольный набор прямоугольников разного цвета (рис. 2.6). Для сохранения результатов используется масштабируемый векторный формат SVG (Scalable Vector Format). Фактически SVG-формат является разновидностью XML-формата, используемого для описания сложной графики в независимом от платформы виде. Более подробные сведения об этом формате можно найти по адресу `http://www.w3c.org/Graphics/SVG`. Для просмотра SVG-файлов следует использовать специализированную программу просмотра Batik компании Apache (`http://xmlgraphics.apache.org/batik`) (рис. 2.7).

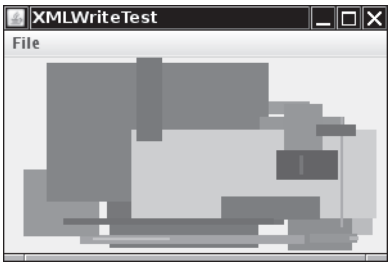


Рис. 2.6. Изображение, которое предстоит сохранить в формате SVG

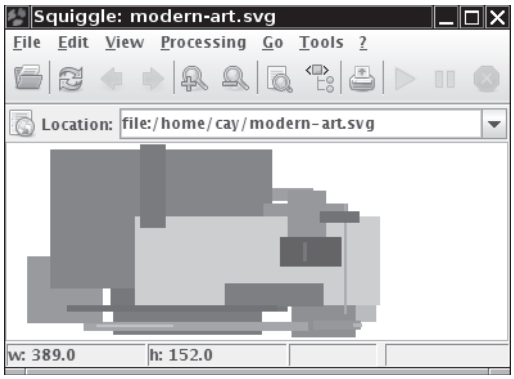


Рис. 2.7. Программа Apache Batik, предназначенная для просмотра SVG-файлов

Мы не будем подробно рассматривать формат SVG. Те, кого заинтересовала эта тема, могут найти прекрасное руководство на Web-сайте компании Adobe. Для нашего примера достаточно лишь знать способ представления набора цветных прямоугольников.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20000802//EN"
  "http://www.w3c.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd">
<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 видно в данном фрагменте кода, каждый прямоугольник описывается узлом `rect`, атрибуты которого задают координаты, ширину, высоту и цвет прямоугольника. Цвет выражается в виде RGB-значения, представленного в шестнадцатеричном формате.



В SVG-формате широко используются атрибуты. Причем некоторые из них имеют очень сложную структуру, как, например, атрибут, применяемый для описания контура:

```
<path d="M 100 100 L 300 100 L 200 300 z">
```

Здесь `M` означает команду `moveto` (перейти), `L` — `lineto` (линия), `z` — `closepath` (замкнуть контур). Наверное, создатели этого формата данных не совсем доверяли формату XML. Очевидно, что вместо таких сложных атрибутов следовало бы использовать элементы.

API javax.xml.parsers.DocumentBuilder 1.4

- `Document newDocument()`
  Возвращает пустой документ.

API org.w3c.dom.Document 1.4

- `Element createElement(String name)`
  Возвращает элемент с заданным именем.
- `Text createTextNode(String data)`
  Возвращает текстовый узел с заданными данными.

API org.w3c.dom.Node 1.4

- `void appendChild(Node child)`
  Добавляет узел к списку узлов, дочерних по отношению к данному. Возвращает включенный узел.

API org.w3c.dom.Element 1.4

- `void setAttribute(String name, String value)`
  Устанавливает для атрибута с заданным именем указанное значение.
- `void setAttributeNS(String uri, String qname, String value)`
  Устанавливает значение атрибута, заданного посредством URI пространства имен и определенного имени.

  Параметры:

`uri`	URI пространства имен или значение `null`.
`qname`	Определенное имя. Если оно имеет префикс-псевдоним, то параметр `uri` должен быть отличным от `null`.
`value`	Значение атрибута.

API javax.xml.transform.TransformerFactory 1.4

- `static TransformerFactory newInstance()`
  Возвращает экземпляр класса `TransformerFactory`.
- `Transformer newTransformer()`
  Возвращает экземпляр класса `Transformer`, который выполняет идентичное преобразование, т.е. преобразование, не предполагающее никаких действий.

API 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`.

Ниже перечислены свойства, применяемые наиболее часто.

doctype-public	Идентификатор PUBLIC, используемый в декларации DOCTYPE.
doctype-system	Идентификатор SYSTEM, используемый в декларации DOCTYPE.
indent	"yes" или "no".
method	"xml", "html", "text" или специальная строка.

- void transform(Source from, Result to)
  Выполняет преобразование XML-документа.

API **javax.xml.transform.dom.DOMSource 1.4**

- DOMSource(Node n)
  Создает источник данных на основе заданного узла. Обычно n представляет собой узел документа.

API **javax.xml.transform.stream.StreamResult 1.4**

- StreamResult(File f)
- StreamResult(OutputStream out)
- StreamResult(Writer out)
- StreamResult(String systemID)
  Создает поток на основе указанного файла, потока, объекта Writer или системного идентификатора (как правило, это относительный или абсолютный URL).

Создание XML-документа посредством StAX

В предыдущем разделе вы видели, как создается XML-документ посредством написания DOM-дерева. Если DOM-дерево вы больше нигде не используете, то этот вариант будет не очень эффективным.

StAX API позволяет сразу написать XML-дерево. Для этого нужно создать XMLStreamWriter из OutputStream, как показано ниже:

```
XMLOutputFactory factory = XMLOutputFactory.newInstance();
XMLStreamWriter writer = factory.createXMLStreamWriter(out);
```

Чтобы создать XML-заголовок, вызовите следующий метод:

```
writer.writeStartDocument()
```

После этого вызовите метод:

```
writer.writeStartElement(name);
```

Добавьте атрибуты, вызвав такой метод:

```
writer.writeAttribute(name, value);
```

Теперь можно добавить дочерние элементы, снова вызвав метод writeStartElement, или записать символы посредством метода

```
writer.writeCharacters(text);
```

После того как вы напишите все дочерние узлы, вызовите метод

```
writer.writeEndElement();
```

Этот метод закроет текущий элемент.

Чтобы написать элемент без потомков (например,), следует вызвать метод

```
writer.writeEmptyElement(name);
```

В конце документа вызовите метод

```
writer.writeEndDocument();
```

Этот метод закроет все открытые элементы.

Как и при использовании подхода с DOM/XSLT, вам не следует беспокоиться о пропуске символов в значениях атрибутов и данных символов. Однако в этом случае существует вероятность того, что вы создадите плохо сформированный XML, например, документ с множеством корневых узлов. Кроме того, текущая версия StAX не поддерживает создание вывода с отступами.

Программа, код которой показан в листинге 2.10, демонстрирует пример применения каждого из этих подходов для написания XML.

Листинг 2.10. Содержимое файла `XMLWriteTest.java`

```
1.  import java.awt.*;
2.  import java.awt.geom.*;
3.  import java.io.*;
4.  import java.util.*;
5.  import java.awt.event.*;
6.  import javax.swing.*;
7.  import javax.xml.parsers.*;
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.  * Данная программа показывает пример написания XML-файла. Она сохраняет
16.  * файл формата SVG с описанием современных приемов рисования.
17.  * @version 1.10 2004-09-04
18.  * @author Cay Horstmann
19.  */
20.
21. public class XMLWriteTest
22. {
23.    public static void main(String[] args)
24.    {
25.       EventQueue.invokeLater(new Runnable()
26.          {
27.             public void run()
28.             {
29.                XMLWriteFrame frame = new XMLWriteFrame();
30.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
31.                frame.setVisible(true);
32.             }
33.          });
34.    }
35. }
36.
37. /**
38.  * Фрейм с компонентом для отображения современных приемов рисования.
39.  */
```

```
40.  class XMLWriteFrame extends JFrame
41.  {
42.     public XMLWriteFrame()
43.     {
44.        setTitle("XMLWriteTest");
45.        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
46.
47.        chooser = new JFileChooser();
48.
49.        // Добавление компонента в фрейм.
50.
51.        comp = new RectangleComponent();
52.        add(comp);
53.
54.        // Задание строки меню.
55.
56.        JMenuBar menuBar = new JMenuBar();
57.        setJMenuBar(menuBar);
58.
59.        JMenu menu = new JMenu("File");
60.        menuBar.add(menu);
61.
62.        JMenuItem newItem = new JMenuItem("New");
63.        menu.add(newItem);
64.        newItem.addActionListener(new ActionListener()
65.           {
66.              public void actionPerformed(ActionEvent event)
67.              {
68.                 comp.newDrawing();
69.              }
70.           });
71.
72.        JMenuItem saveItem = new JMenuItem("Save with DOM/XSLT");
73.        menu.add(saveItem);
74.        saveItem.addActionListener(new ActionListener()
75.           {
76.              public void actionPerformed(ActionEvent event)
77.              {
78.                 try
79.                 {
80.                    saveDocument();
81.                 }
82.                 catch (Exception e)
83.                 {
84.                    JOptionPane.showMessageDialog(XMLWriteFrame.this, e.toString());
85.                 }
86.              }
87.           });
88.
89.        JMenuItem saveStAXItem = new JMenuItem("Save with StAX");
90.        menu.add(saveStAXItem);
91.        saveStAXItem.addActionListener(new ActionListener()
92.           {
93.              public void actionPerformed(ActionEvent event)
94.              {
95.                 try
96.                 {
```

```
97.                saveStAX();
98.             }
99.             catch (Exception e)
100.            {
101.               JOptionPane.showMessageDialog(XMLWriteFrame.this, e.toString());
102.            }
103.         }
104.      });
105.
106.   JMenuItem exitItem = new JMenuItem("Exit");
107.   menu.add(exitItem);
108.   exitItem.addActionListener(new ActionListener()
109.      {
110.         public void actionPerformed(ActionEvent event)
111.         {
112.            System.exit(0);
113.         }
114.      });
115. }
116.
117. /**
118.  * Сохранение рисунка в формате SVG посредством DOM/XSLT.
119.  */
120. public void saveDocument() throws TransformerException, IOException
121. {
122.    if (chooser.showSaveDialog(this) != JFileChooser.APPROVE_OPTION) return;
123.    File f = chooser.getSelectedFile();
124.    Document doc = comp.buildDocument();
125.    Transformer t = TransformerFactory.newInstance().newTransformer();
126.    t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM,
127.       "http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd");
128.    t.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC, "-//W3C//DTD SVG 20000802//EN");
129.    t.setOutputProperty(OutputKeys.INDENT, "yes");
130.    t.setOutputProperty(OutputKeys.METHOD, "xml");
131.    t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
132.    t.transform(new DOMSource(doc), new StreamResult(new FileOutputStream(f)));
133. }
134.
135. /**
136.  * Сохранение рисунка в формате SVG посредством StAX.
137.  */
138. public void saveStAX() throws FileNotFoundException, XMLStreamException
139. {
140.    if (chooser.showSaveDialog(this) != JFileChooser.APPROVE_OPTION) return;
141.    File f = chooser.getSelectedFile();
142.    XMLOutputFactory factory = XMLOutputFactory.newInstance();
143.    XMLStreamWriter writer = factory.createXMLStreamWriter(new FileOutputStream(f));
144.    comp.writeDocument(writer);
145.    writer.close();
146. }
147.
148. public static final int DEFAULT_WIDTH = 300;
149. public static final int DEFAULT_HEIGHT = 200;
150.
151. private RectangleComponent comp;
152. private JFileChooser chooser;
153. }
```

```
154. /**
155.  * Компонент, показывающий набор цветных прямоугольников.
156.  */
157. class RectangleComponent extends JComponent
158. {
159.    public RectangleComponent()
160.    {
161.       rects = new ArrayList<Rectangle2D>();
162.       colors = new ArrayList<Color>();
163.       generator = new Random();
164.
165.       DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
166.       try
167.       {
168.          builder = factory.newDocumentBuilder();
169.       }
170.       catch (ParserConfigurationException e)
171.       {
172.          e.printStackTrace();
173.       }
174.    }
175.
176.    /**
177.     * Создание нового рисунка случайной формы.
178.     */
179.    public void newDrawing()
180.    {
181.       int n = 10 + generator.nextInt(20);
182.       rects.clear();
183.       colors.clear();
184.       for (int i = 1; i <= n; i++)
185.       {
186.          int x = generator.nextInt(getWidth());
187.          int y = generator.nextInt(getHeight());
188.          int width = generator.nextInt(getWidth() - x);
189.          int height = generator.nextInt(getHeight() - y);
190.          rects.add(new Rectangle(x, y, width, height));
191.          int r = generator.nextInt(256);
192.          int g = generator.nextInt(256);
193.          int b = generator.nextInt(256);
194.          colors.add(new Color(r, g, b));
195.       }
196.       repaint();
197.    }
198.
199.    public void paintComponent(Graphics g)
200.    {
201.       if (rects.size() == 0) newDrawing();
202.       Graphics2D g2 = (Graphics2D) g;
203.       // Рисование всех прямоугольников.
204.       for (int i = 0; i < rects.size(); i++)
205.       {
206.          g2.setPaint(colors.get(i));
207.          g2.fill(rects.get(i));
208.       }
209.    }
210.
```

```
211.   /**
212.    * Создание SVG-документа с текущим рисунком.
213.    * @return DOM-дерево SVG-документа
214.    */
215.   public Document buildDocument()
216.   {
217.     Document doc = builder.newDocument();
218.     Element svgElement = doc.createElement("svg");
219.     doc.appendChild(svgElement);
220.     svgElement.setAttribute("width", "" + getWidth());
221.     svgElement.setAttribute("height", "" + getHeight());
222.     for (int i = 0; i < rects.size(); i++)
223.     {
224.       Color c = colors.get(i);
225.       Rectangle2D r = rects.get(i);
226.       Element rectElement = doc.createElement("rect");
227.       rectElement.setAttribute("x", "" + r.getX());
228.       rectElement.setAttribute("y", "" + r.getY());
229.       rectElement.setAttribute("width", "" + r.getWidth());
230.       rectElement.setAttribute("height", "" + r.getHeight());
231.       rectElement.setAttribute("fill", colorToString(c));
232.       svgElement.appendChild(rectElement);
233.     }
234.     return doc;
235.   }
236.
237.   /**
238.    * Писатели SVG-документа с текущим рисунком.
239.    * @param писатель документа
240.    */
241.   public void writeDocument(XMLStreamWriter writer) throws XMLStreamException
242.   {
243.     writer.writeStartDocument();
244.     writer.writeDTD("<!DOCTYPE svg PUBLIC \"-//W3C//DTD SVG 20000802//EN\" " +
245.       "\"http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd\">");
246.     writer.writeStartElement("svg");
247.     writer.writeAttribute("width", "" + getWidth());
248.     writer.writeAttribute("height", "" + getHeight());
249.     for (int i = 0; i < rects.size(); i++)
250.     {
251.       Color c = colors.get(i);
252.       Rectangle2D r = rects.get(i);
253.       writer.writeEmptyElement("rect");
254.       writer.writeAttribute("x", "" + r.getX());
255.       writer.writeAttribute("y", "" + r.getY());
256.       writer.writeAttribute("width", "" + r.getWidth());
257.       writer.writeAttribute("height", "" + r.getHeight());
258.       writer.writeAttribute("fill", colorToString(c));
259.     }
260.     writer.writeEndDocument(); // Закрываем svg-элемент.
261.   }
262.
263.   /**
264.    * Преобразование цвета в шестнадцатеричное значение.
265.    * @param c цвет
266.    * @return строка в виде #rrggbb
267.    */
```

```
268.   private static String colorToString(Color c)
269.   {
270.      StringBuffer buffer = new StringBuffer();
271.      buffer.append(Integer.toHexString(c.getRGB() & 0xFFFFFF));
272.      while (buffer.length() < 6)
273.         buffer.insert(0, '0');
274.      buffer.insert(0, '#');
275.      return buffer.toString();
276.   }
277.
278.   private ArrayList<Rectangle2D> rects;
279.   private ArrayList<Color> colors;
280.   private Random generator;
281.   private DocumentBuilder builder;
282. }
```

API javax.xml.stream.XMLOutputFactory

- `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)`

  Создает писатель, который осуществляет запись в данный поток, в писатель или в результаты JAXP.

API javax.xml.stream.XMLStreamWriter

- `void writeStartDocument()`
- `void writeStartDocument(String xmlVersion)`
- `void writeStartDocument(String encoding, String xmlVersion)`

  Записывает инструкцию обработки XML в верхней части документа. Обратите внимание, что параметр `encoding` используется только для записи атрибута. Он не определяет шифрование символов в выходных данных.
- `void setDefaultNamespace(String namespaceURI)`
- `void setPrefix(String prefix, String namespaceURI)`

  Задает пространство имен по умолчанию или пространство имен, связанное с префиксом. Объявление действительно для текущего элемента или, если ни одного элемента не было записано — для корня документа.
- `void writeStartElement(String localName)`
- `void writeStartElement(String namespaceURI, String localName)`

  Записывает начальный дескриптор, заменяя `namespaceURI` связанным префиксом.
- `void writeEndElement()`

  Закрывает текущий элемент.

- `void writeEndDocument()`
  Закрывает все открытые элементы.
- `void writeEmptyElement(String localName)`
- `void writeEmptyElement(String namespaceURI, String localName)`
  Записывает самозакрывающийся дескриптор, заменяя `namespaceURI` связанным префиксом.
- `void writeAttribute(String localName, String value)`
- `void writeAttribute(String namespaceURI, String localName, String value)`
  Записывает атрибут для текущего элемента, заменяя `namespaceURI` связанным префиксом.
- `void writeCharacters(String text)`
  Записывает данные символов.
- `void writeCData(String text)`
  Записывает блок `CDATA`.
- `void writeDTD(String dtd)`
  Записывает строку `dtd`, которая должна содержать объявление `DOCTYPE`.
- `void writeComment(String comment)`
  Записывает комментарий.
- `void close()`
  Закрывает писатель.

XSLT-преобразования

Механизм XSLT (XSL Transformation — преобразования XSL) позволяет определять правила преобразования XML-документов в другие форматы, такие как простой текст, XHTML или любой другой XML-формат. XSLT обычно используется для перевода из одного XML-формата, который может прочитать компьютер, в другой формат, либо же для перевода XML в другой формат представления, доступный для понимания людьми.

Вам необходимо создать таблицу стилей XSLT, которая смогла бы описывать преобразование XML-документов в некоторый другой формат. Процессор XSLT читает XML-документ и таблицу стилей, и выдает желаемый результат (рис. 2.8).

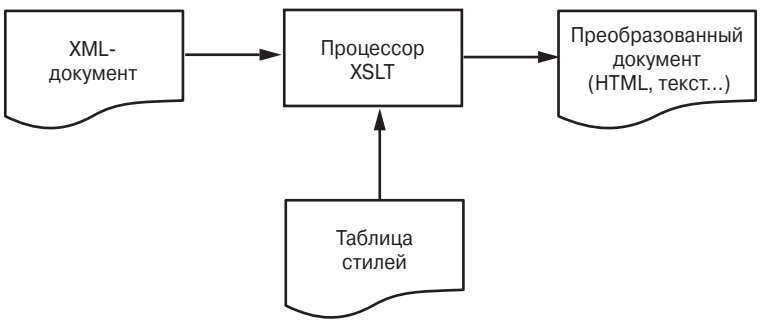


Рис. 2.8. Применение XSLT-преобразования

Рассмотрим пример преобразования XML-файла с записями о сотрудниках в HTML-документ. Исходные данные в формате XML приведены ниже.

```
<staff>
  <employee>
    <name>Carl Cracker</name>
    <salary>75000</salary>
    <hiredate year="1987" mobth="12" day="15"/>
  </employee>
  <employee>
    <name>Harry Cracker</name>
    <salary>50000</salary>
    <hiredate year="1989" mobth="10" day="1"/>
  </employee>
  <employee>
    <name>Tony Tester</name>
    <salary>40000</salary>
    <hiredate year="1990" mobth="3" day="15"/>
  </employee>
</staff>
```

На его основании желательно получить HTML-таблицу:

```
<table border="1">
<tr>
<td>Carl Cracker</td><td>$75000.0</td><td>1987-12-15</td>
</tr>
<tr>
<td>Harry Cracker</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 XSLT имеет чрезвычайно сложную структуру, и ее описанию посвящено довольно много книг. Ограничимся обсуждением лишь некоторых элементов, которые используются в рассматриваемом здесь примере. Более подробную информацию по этой теме можно найти в книге Дона Бокса (Don Box) и др. *Essential XML* (Addison-Wesley Professional, 2000 г.), а описание спецификации XSLT находится по адресу `http://www.w3c.org/TR/xslt`.

Ниже приведена 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"/>
  шаблон_1
  шаблон_2
  . . .
</xsl:stylesheet>
```

В данном примере элемент `xsl:output` содержит атрибут `method`, для которого указано значение `"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 /staff/employee нужно выполнить описанные ниже действия.

1. Включить дескриптор строки <tr>.
2. Применять этот шаблон для узлов, дочерних по отношению к данному.
3. После обработки всех дочерних узлов включить закрывающий дескриптор </tr>.

Иначе говоря, этот шаблон генерирует вокруг каждой записи о сотрудниках дескрипторы строк HTML-таблицы.

Процессор XSLT начинает обработку с проверки корневого элемента. Если узел соответствует одному из шаблонов, этот шаблон немедленно применяется. (При совпадении нескольких шаблонов используется шаблон с наибольшей степенью соответствия. Более подробные сведения приведены в спецификации XSLT – ее вы можете найти по адресу http://www.w3.org/TR/xslt.) Если соответствие с шаблоном не найдено, то XSLT-процессор выполняет заданное по умолчанию действие. Для текстовых узлов по умолчанию в выходной поток записывается содержимое, а для элементов выходной поток не создается, но продолжается обработка дочерних узлов.

Ниже показан шаблон преобразования узлов name в файле с данными о сотрудниках:

```
<xsl:template match="/staff/employee/name">
   <td><xsl:apply-templates/></td>
</xsl:template>
```

Как видите, шаблон создает дескрипторы <td>...</td> и предлагает XSLT-процессору рекурсивно обработать дочерние узлы элемента name. Этот элемент имеет только один дочерний текстовый узел. Когда процессор обрабатывает узел, он возвращает его текстовое содержимое (при условии, что никакого другого шаблона не задано).

Для копирования значений атрибутов в выходной поток 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 в соответствии с этим шаблоном будут сгенерированы перечисленные ниже компоненты.

- Дескриптор <td>.
- Значение атрибута year.
- Дефис.
- Значение атрибута month.
- Дефис.
- Значение атрибута day.
- Дефис.
- Дескриптор </td>.

Выражение xsl:value-of вычисляет строковое значение набора узлов, указанного с помощью XPath-значения атрибута select. В этом случае путь вычисляется относительно текущего узла. Набор узлов преобразуется в строку путем конкатенации

строковых значений всех узлов. Строковым значением атрибута является его значение, строковым значением текстового узла — его содержимое, а строковым значением элемента — конкатенация строковых значений его дочерних узлов (но не атрибутов).

В листинге 2.11 приведен шаблон, предназначенный для преобразования XML-файла с записями о сотрудниках в HTML-таблицу.

В листинге 2.12 показаны шаблоны для преобразования того же XML-файла в текст в знакомом вам формате файла свойств.

```
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
```

В данном примере используется функция `position()`, которая сообщает о расположении текущего узла по отношению к родительскому узлу. Для получения другого выходного формата достаточно изменить таблицу стилей. Таким образом, XML-документ можно спокойно применять для представления информации, не заботясь о том, в каком формате она требуется приложению. Для генерации данных в нужном формате достаточно воспользоваться XSLT.

XSLT-преобразования можно легко организовать на языке Java. Для этого нужно для каждой таблицы стилей создать фабрику преобразований, затем получить объект `Transformer` и передать ему преобразуемые данные:

```
File styleSheet = new File(filename);
StreamSource styleSource = new StreamSource(styleSheet);
Transformer t = TransformerFactory.newInstance().newTransformer(styleSource);
t.transform(source, result);
```

Параметры метода `transform()` представляют собой экземпляры классов, которые реализуют интерфейсы `Source` и `Result`. Интерфейс источника данных `Source` имеет три реализации:

```
DOMSource
SAXSource
StreamSource
```

Источник данных `StreamSource` можно создать на основе файла, потока, объекта `Reader` или URL, а источник `DOMSource` — на основе узла, принадлежащего дереву DOM. Например, в предыдущем разделе использовалось следующее преобразование:

```
t.transform(new DOMSource(doc), result);
```

В данном примере применяется другой подход. Вместо использования существующего XML-файла создается объект чтения XML-данных, который имитирует SAX-анализ с генерацией соответствующих SAX-событий. Фактически используемый здесь объект считывает обычный текстовый файл:

```
Carl Cracker|75000.0|1987|12|15
Harry Hacker|50000.0|1989|10|1
Tony Tester|40000.0|1990|3|15
```

По мере обработки входного потока объект чтения генерирует SAX-события. Ниже представлен фрагмент метода `parse()` класса `EmployeeReader`, реализующего интерфейс `XMLReader`.

```
String rootElement = "staff";
AttributesImpl attributes = new AttributesImpl();

if (handler == null)
   throw new SAXException("No content handler");

handler.startDocument();
handler.startElement("", rootElement, rootElement, atts);
String line;
while ((line = in.readLine()) != null)
{
   handler.startElement("", "employee", "employee", atts);
   StringTokenizer t = new StringTokenizer(line, "|");

   handler.startElement("", "name", "name", atts);
   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 данных `SAXSource` для преобразования создается на основе объекта чтения XML-данных:

```
t.transform(new SAXSource(new EmployeeReader(),
       new InputSource(new FileInputStream(filename))),
       result);
```

Этот способ как нельзя лучше подходит для преобразования уже имеющихся данных в формат XML. Конечно, для большинства XSLT-приложений входные данные имеют тип XML, что позволяет использовать метод `transform()` так, как показано ниже:

```
t.transform(new StreamSource(file), result);
```

Результатом такого преобразования будет объект одного из трех классов, которые реализуют интерфейс `Result`:

```
DOMResult
SAXResult
StreamResult
```

Для сохранения результата в виде DOM-дерева нужно использовать объект `DocumentBuilder`. С его помощью генерируется новый документ и помещается в оболочку `DOMResult`:

```
Document doc = builder.newDocument();
t.transform(source, new DOMResult(doc));
```

Для сохранения выходного потока в виде файла следует использовать объект `StreamResult`:

```
t.transform(source, new StreamResult(file));
```

В листинге 2.13 приведен исходный код программы.

Листинг 2.11. Содержимое файла `makehtml.xsl`

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet
   xmlns:xsl="http://www.w3c.org/1999/XSL/Transform"
   version="1.0">

   <xsl:output method="html"/>

   <xsl:template match="/staff">
      <table border="1"><xsl:apply-templates/></table>
   </xsl:template>

   <xsl:template match="/staff/employee">
      <tr><xsl:apply-templates/></tr>
   </xsl:template>

   <xsl:template match="/staff/employee/name">
      <td><xsl:apply-templates/></td>
   </xsl:template>

   <xsl:template match="/staff/employee/salary">
      <td>$<xsl:apply-templates/></td>
   </xsl:template>

   <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>

</xsl:stylesheet>
```

Листинг 2.12. Содержимое файла `makeprop.xsl`

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

<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">

   <xsl:output method="text"/>
   <xsl:template match="/staff/employee">
   employee.<xsl:value-of
   select="position()"/>.name=<xsl:value-of
   select="name/text()"/>
   employee.<xsl:value-of
   select="position()"/>.salary=<xsl:value-of
   select="salary/text()"/>
   employee.<xsl:value-of
   select="position()"/>.hiredate=<xsl:value-of
   select="hiredate/@year"/>-<xsl:value-of
   select="hiredate/@month"/>-<xsl:value-of
   select="hiredate/@day"/>
</xsl:template>

</xsl:stylesheet>
```

Листинг 2.13. Содержимое файла `TransformTest.java`

```
1.  import java.io.*;
2.  import java.util.*;
3.  import javax.xml.transform.*;
4.  import javax.xml.transform.sax.*;
5.  import javax.xml.transform.stream.*;
6.  import org.xml.sax.*;
7.  import org.xml.sax.helpers.*;
8.  /**
9.   * Пример XSLT-преобразования записей о сотрудниках.
10.  * Записи хранятся в файле employee.dat и читаются
11.  * как XML-данные. С помощью таблиц стилей данные
12.  * преобразуются в формат HTML и в текстовый формат.
13.  * В командной строке необходимо указать требуемую таблицу
14.  * стилей. Например, для получения данных в HTML-формате
15.  * программа должна быть вызвана следующим образом:
16.  * java TransformTest makeprop.xsl
17.  * @version 1.01 2007-06-25
18.  * @author Cay Horstmann
19.  */
20. public class TransformTest
21. {
22.    public static void main(String[] args) throws Exception
23.    {
24.       String filename;
25.       if (args.length > 0) filename = args[0];
26.       else filename = "makehtml.xsl";
27.       File styleSheet = new File(filename);
28.       StreamSource styleSource = new StreamSource(styleSheet);
29.
30.       Transformer t = TransformerFactory.newInstance().
31.          newTransformer(styleSource);
32.       t.setOutputProperty(OutputKeys.INDENT, "yes");
33.       t.setOutputProperty(OutputKeys.METHOD, "xml");
34.       t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
35.
36.       t.transform(new SAXSource(new EmployeeReader(),
37.          new InputSource(new FileInputStream("employee.dat"))),
38.          new StreamResult(System.out));
39.    }
40. }
41. /**
42.  * Этот класс считывает данные из текстового файла employee.dat
43.  * и генерирует события SAX-анализатора. В результате разбор
44.  * текстового файла происходит так же, как и анализ XML-файла.
45.  */
46. class EmployeeReader implements XMLReader
47. {
48.    public void parse(InputSource source)
49.       throws IOException, SAXException
50.    {
51.       InputStream stream = source.getByteStream();
52.       BufferedReader in = new BufferedReader(new InputStreamReader(stream));
53.       String rootElement = "staff";
54.       AttributesImpl atts = new AttributesImpl();
55.
```

```
56.       if (handler == null)
57.         throw new SAXException("No content handler");
58.       handler.startDocument();
59.       handler.startElement("", rootElement, rootElement, atts);
60.       String line;
61.       while ((line = in.readLine()) != null)
62.       {
63.         handler.startElement("", "employee", "employee", atts);
64.         StringTokenizer t = new StringTokenizer(line, "|");
65.
66.         handler.startElement("", "name", "name", atts);
67.         String s = t.nextToken();
68.         handler.characters(s.toCharArray(), 0, s.length());
69.         handler.endElement("", "name", "name");
70.
71.         handler.startElement("", "salary", "salary", atts);
72.         s = t.nextToken();
73.         handler.characters(s.toCharArray(), 0, s.length());
74.         handler.endElement("", "salary", "salary");
75.
76.         atts.addAttribute("", "year", "year", "CDATA", t.nextToken());
77.         atts.addAttribute("", "month", "month", "CDATA", t.nextToken());
78.         atts.addAttribute("", "day", "day", "CDATA", t.nextToken());
79.         handler.startElement("", "hiredate", "hiredate", atts);
80.         handler.endElement("", "hiredate", "hiredate");
81.         atts.clear();
82.
83.         handler.endElement("", "employee", "employee");
84.       }
85.       handler.endElement("", rootElement, rootElement);
86.       handler.endDocument();
87.     }
88.     public void setContentHandler(ContentHandler newValue)
89.       {
90.         handler = newValue;
91.       }
92.     public ContentHandler getContentHandler()
93.       {
94.         return handler;
95.       }
96.
97.     // При вызове следующих методов
98.     // никакие действия не выполняются.
99.     public void parse(String systemId)
100.      throws IOException, SAXException
101.    {
102.    }
103.
104.    public void setErrorHandler(ErrorHandler handler)
105.    {
106.    }
107.
108.    public ErrorHandler getErrorHandler()
109.    {
110.      return null;
111.    }
112.
```

```
113.    public void setDTDHandler(DTDHandler handler)
114.    {
115.    }
116.
117.    public DTDHandler getDTDHandler()
118.    {
119.       return null;
120.    }
121.
122.    public void setEntityResolver(EntityResolver resolver)
123.    {
124.    }
125.
126.    public EntityResolver getEntityResolver()
127.    {
128.       return null;
129.    }
130.
131.    public void setProperty(String name, Object value)
132.    {
133.    }
134.
135.    public Object getProperty(String name)
136.    {
137.       return null;
138.    }
139.
140.    public void setFeature(String name, boolean value)
141.    {
142.    }
143.
144.    public boolean getFeature(String name)
145.    {
146.       return false;
147.    }
148.    private ContentHandler handler;
149. }
```

API javax.xml.transform.TransformerFactory 1.4

- `Transformer newTransformer(Source styleSheet)`

  Возвращает экземпляр класса `Transformer`, который считывает таблицу стилей из указанного источника.

API javax.xml.transform.stream.StreamSource 1.4

- `StreamSource(File f)`
- `StreamSource(InputStream in)`
- `StreamSource(Reader in)`
- `StreamSource(String systemID)`

  Создает поток-источник входных данных на базе файла, потока, объекта `Reader` или системного идентификатора. Обычно в роли системного идентификатора выступает относительный или абсолютный URL.

API javax.xml.transform.sax.SAXSource 1.4

- `SAXSource(XMLReader reader, InputSource source)`
  Создает на базе указанного входного потока SAX-источник данных и использует для анализа заданный объект `Reader`.

API org.xml.sax.XMLReader 1.4

- `void setContentHandler(ContentHandler handler)`
  Устанавливает обработчик, который уведомляется о событиях, возникающих при анализе.
- `void parse(InputSource source)`
  Анализирует данные из заданного источника и передает возникающие события обработчику содержимого.

API javax.xml.transform.dom.DOMResult 1.4

- `DOMResult(Node n)`
  Создает источник данных на основе заданного узла. Обычно в качестве параметра указывается узел документа.

API org.xml.sax.helpers.AttributesImpl 1.4

- `void addAttribute(String uri, String lname, String qname, String type, String value)`
  Добавляет атрибут к набору.

  Параметры:

`uri`	URI пространства имен.
`lname`	Локальное имя без префикса-псевдонима.
`qname`	Определенное имя с префиксом-псевдонимом.
`type`	Один из следующих типов: `"CDATA"`, `"ID"`, `"IDREF"`, `"IDREFS"`, `"NMTOKEN"`, `"NMTOKENS"`, `"ENTITY"`, `"ENTITIES"` или `"NOTATION"`.
`value`	Значение атрибута.

- `void clear()`

  Удаляет все атрибуты из набора.

Этим примером мы завершаем обсуждение вопроса поддержки XML в библиотеке Java. Теперь у вас должно быть хорошее представление о том, какими возможностями обладает XML, в частности, автоматизированный разбор и проверка, а также мощный механизм преобразований. Естественно, вся эта технология сможет заработать, если вы тщательно подготовите свои форматы XML. Вы должны убедиться в том, что форматы способны удовлетворить всем существующим бизнес-требованиям, что они являются стабильными с течением времени и что деловые партнеры готовым принимать ваши XML-документы. Решение этих вопросов может оказаться гораздо сложнее, чем работа с анализаторами, DTD или преобразованиями.

В следующей главе мы рассмотрим вопросы программирования для сети на платформе Java. Сначала мы поговорим об основах сетевых сокетов, а затем перейдем к высокоуровневым протоколам для электронной почты и World Wide Web.

Глава 3

РАБОТА В СЕТИ

В ЭТОЙ ГЛАВЕ...

- Подключение к серверу
- Реализация серверов
- Прерываемые сокеты
- Отправка электронной почты
- Создание соединений с URL

Эта глава начинается с описания основ организации сетевых соединений, а затем продолжается рассмотрением Java-программ, позволяющих устанавливать соединения с серверами. Вы узнаете, как производится реализация сетевых клиентов и серверов. Завершается глава рассмотрением вопроса передачи почтовых сообщений из программы на Java и сбора информации с Web-сервера.

Подключение к серверу

Перед созданием первой сетевой программы рекомендуется познакомиться с прекрасным инструментом отладки сетевых программ, а именно — с утилитой `telnet`. Учтите, что она не является обязательным компонентом операционной системы, поэтому проверьте, установлена ли она на вашем компьютере. Если утилиту `telnet` невозможно запустить из командной строки, еще раз запустите программу инсталляции и выберите ее в списке предлагаемых для установки компонентов.



Вместе с ОС Windows Vista устанавливается и утилита `telnet`, однако по умолчанию она не активизирована. Чтобы активизировать ее, откройте панель управления, перейдите в раздел Программы, щелкните на ссылке Добавление или удаление компонентов Windows и отметьте флажок Клиент Telnet. К тому же, брандмауэр Windows блокирует некоторые сетевые порты, которыми мы будем пользоваться в этой главе; чтобы разблокировать порты, вы должны обладать полномочиями администратора.

Утилита `telnet` может использоваться не только для соединения с удаленным компьютером. Посредством ее вы также можете взаимодействовать с различными сетевыми службами. Ниже приводится один из примеров необычного использования этой утилиты. Введите в командной строке команду

```
telnet time-A.timefreq.bldrdoc.gov 13
```

На рис. 3.1 показан пример ответной реакции сервера, которая в режиме командной строки будет иметь следующий вид:

```
53221 04-08-04 02:19:40 50 0 0 513.0 UTC(NIST) *
```

Что же произошло в этом случае? Утилита `telnet` подключилась к серверу службы времени, 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) в Боулдере, штат Колорадо, США. 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 договоренности сервер службы времени всегда связан с портом 13.



В терминологии, используемой для описания сетевых технологий, порт — это не какое-то физическое устройство, а абстрактное понятие, которое используется для описания организации соединения между сервером и клиентом (рис. 3.2).

Программное обеспечение сервера постоянно работает на удаленном компьютере и ожидает поступления сетевого трафика по порту 13. При получении операционной системой на удаленном компьютере сетевого пакета с запросом на подключение с использованием порта 13 активизируется процесс сервера и устанавливается соединение. Такое соединение может быть прервано одним из его участников.

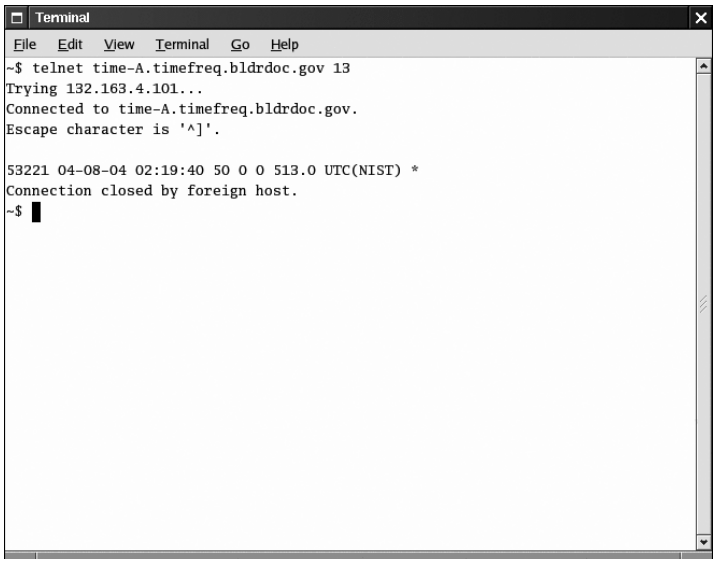


Рис. 3.1. Результат работы службы времени

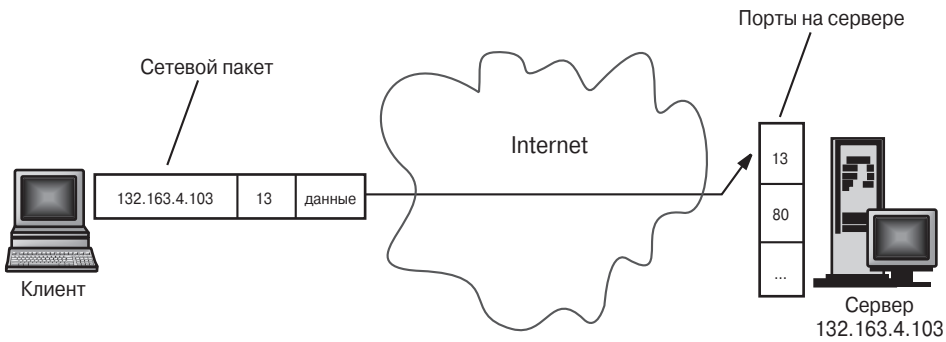


Рис. 3.2. Схема взаимодействия клиента с сервером по определенному порту

После запуска утилиты `telnet` с параметром `time-A.timefreq.bldrdoc.gov` и портом `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 по порту 13. После установления соединения программа на удаленном компьютере передает обратно строку с данными, а затем закрывает соединение. В общем случае клиенты и серверы до закрытия соединения могут участвовать в более сложных диалогах.

Вот еще один более интересный эксперимент. Выполните перечисленные ниже действия.

1. Подключитесь к серверу `java.sun.com` по порту `80`.
2. Введите строку `GET / HTTP/1.0` *точно в показанном здесь виде без использования клавиши* <Backspace>.
3. Теперь два раза нажмите клавишу <Enter>.

На рис. 3.3 показана ответная реакция сервера в окне утилиты `telnet`. Она имеет уже знакомый нам вид страницы текста в формате HTML, а именно — исходной Web-страницы компании Sun, посвященной технологиям Java.

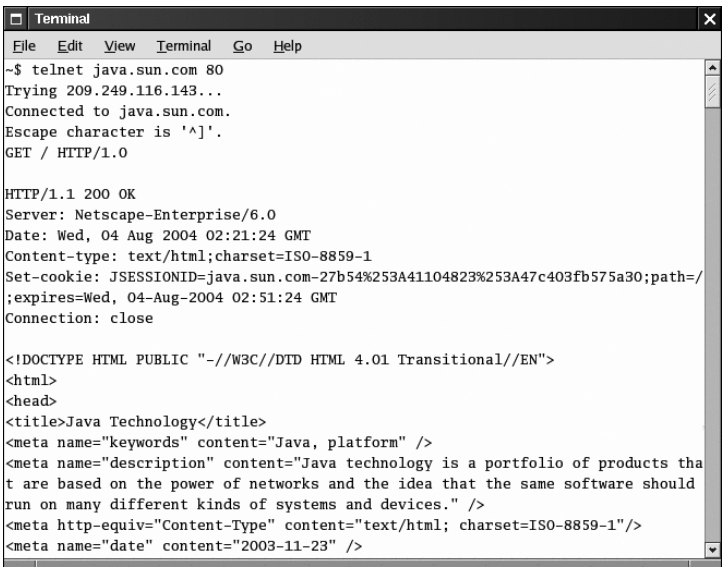


Рис. 3.3. Использование утилиты telnet для доступа к HTTP-порту

Именно так обычный Web-браузер получает все Web-страницы. Для запроса Web-страниц на сервере он применяет HTTP. Естественно, браузер отображает информацию в виде, гораздо более удобном для чтения.



Если вы попытаетесь проделать это с Web-сервером, на котором размещено множество доменов с одинаковым IP-адресом, вы не получите требуемой страницы. (Так бывает с небольшими Web-сайтами, находящимися на одном сервере, таком как `horstmann.com`.) При подключении к такому серверу нужно определить требуемое имя хоста, как показано ниже:

```
GET / HTTP/1.1
Host: horstmann.com
```

Затем нажмите клавишу <Enter> два раза. (Обратите внимание, что HTTP имеет версию 1.1.)

Программа, приведенная в листинге 3.1, выполняет точно такие же действия, т.е. подключается к порту и выводит полученные данные.

Листинг 3.1. Содержимое файла `SocketTest.java`

```
1.  import java.io.*;
2.  import java.net.*;
3.  import java.util.*;
4.
5.  /**
6.   * Данная программа создает сокет для соединения
7.   * с атомными часами в Боулдере, штат Колорадо.
8.   * Она выводит на экран текущее время, полученное с сервера.
```

```
9.    * @version 1.20 2004-08-03
10.   * @author Cay Horstmann
11.   */
12.  public class SocketTest
13.  {
14.    public static void main(String[] args)
15.    {
16.      try
17.      {
18.        Socket s = new Socket("time-A.timefreq.bldrdoc.gov", 13);
19.        try
20.        {
21.          InputStream inStream = s.getInputStream();
22.          Scanner in = new Scanner(inStream);
23.          while (in.hasNextLine())
24.          {
25.            String line = in.nextLine();
26.            System.out.println(line);
27.          }
28.        }
29.        finally
30.        {
31.          s.close();
32.        }
33.      }
34.      catch (IOException e)
35.      {
36.        e.printStackTrace();
37.      }
38.    }
39.  }
```

В данной программе нас больше всего интересуют следующие две строки:

```
Socket s = new Socket("time-A.timefreq.bldrdoc.gov", 13);
InputStream inStream = s.getInputStream();
```

Первая строка позволяет открыть *сокет*. Сокет — это абстрактное понятие, которое поддерживается программным обеспечением, позволяющим организовать операции обмена данными по сети. Конструктору сокета передается адрес удаленного сервера и номер порта. При неудачном соединении генерируется исключение `UnknownHostException`, а при наличии каких-то других проблем — исключение `IOException`. Класс `UnknownHostException` является подклассом `IOException`, поэтому в этом простом примере обрабатывается только исключение суперкласса.

После открытия сокета метод `getInputStream()` класса `java.net.Socket` возвращает объект потока `InputStream`, который можно использовать как любой другой поток. После получения потока программа приступает к выводу каждой строки в стандартный выходной поток. Данный процесс продолжается до тех пор, пока поток не будет закрыт или пока не будет разорвано соединение с сервером.

Эта программа может работать только с очень простыми серверами, например со службой времени. В более сложных случаях клиент посылает запрос на получение данных серверу, а сервер может в течение некоторого времени поддерживать это соединение. Примеры подобного поведения представлены далее в главе.

Класс `Socket` очень удобен в работе, поскольку скрывает все сложные подробности установления сетевого соединения и передачи данных по сети. А пакет `java.net` предоставляет тот же программный интерфейс, который используется для работы с файлами.



В этой книге мы рассмотрим лишь протокол TCP (Transmission Control Protocol — протокол управления передачей). Платформа Java поддерживает также протокол UDP (User Datagram Protocol — протокол дейтаграмм пользователя), который может использоваться для отправки пакетов (они называются также *дейтаграммами* (datagram)) с гораздо меньшими затратами, чем в случае протокола TCP. Недостатком является то, что пакеты не нужно доставлять принимающему приложению в последовательном порядке, и они вообще могут быть потеряны. Получатель сам должен позаботиться о том, чтобы пакеты были организованы в определенном порядке, и сам должен повторно запрашивать передачу отсутствующих пакетов. Протокол UDP хорошо подходит для тех приложений, которые могут обходиться без отсутствующих пакетов — например, в аудио- и видеопотоках, или для продолжительных измерений.

Тайм-ауты сокетов

Чтение из блоков сокетов доступно до тех пор, пока данные являются доступными. Если хост является недосягаемым, ваше приложение будет ожидать в течение длительного периода времени, и все будет зависеть от того, когда операционная система, под управлением которой работает компьютер, определит момент завершения ожидания.

Вы можете решить, какое значение тайм-аута будет разумным для вашего определенного приложения. Затем вызовите метод `setSoTimeout`, чтобы задать значение тайм-аута (в миллисекундах).

```
Socket s = new Socket(. . .);
s.setSoTimeout(10000); // тайм-аут наступит через 10 секунд
```

Если значение тайм-аута было задано для сокета, то все последующие операции чтения и записи будут генерировать исключение `SocketTimeoutException` в момент наступления тайм-аута, прежде чем операция завершит свою работу. Вы можете перехватить исключение и отреагировать на тайм-аут.

```
try
{
   InputStream in = s.getInputStream(); // чтение из in
   . . .
}
catch (InterruptedIOException exception)
{
   реакция на тайм-аут
}
```

Вам придется решить еще одну дополнительную проблему, связанную с тайм-аутами. Конструктор

```
Socket(String host, int port)
```

может удерживать блокировку в течение неопределенного периода времени до тех пор, пока не будет установлено исходное соединение с хостом.

Эту проблему можно преодолеть, если сначала создать несоединенный сокет, а затем соединиться с ним во время тайм-аута:

```
Socket s = new Socket();
s.connect(new InetSocketAddress(host, port), timeout);
```

Если вы хотите позволить пользователям прерывать соединение с сокетом в любой момент времени, прочтите раздел “Прерываемые сокеты” далее в этой главе.

API java.net.Socket 1.0

- `Socket()` 1.1
  Создает сокет, который на данный момент времени не имеет соединения.
- `void connect(SocketAddress address)` 1.4
  Производит подключение сокета к данному адресу.
- `void connect(SocketAddress address, int timeoutInMilliseconds)` 1.4
  Производит подключение сокета к данному адресу или возвращает значение, если временной интервал истек.
- `void setSoTimeout(int timeoutInMilliseconds)` 1.1
  Задает время блокирования чтения запросов на данном сокете. Если наступит тайм-аут, возникнет исключение `InterruptedIOException`.
- `boolean isConnected()` 1.4
  Возвращает значение `true`, если существует соединение с сокетом.
- `boolean isClosed()` 1.4
  Возвращает значение `true`, если соединение с сокетом закрыто.

Internet-адреса

Как правило, вам не нужно излишне беспокоиться об Internet-адресах — числовых адресах хостов, состоящих из четырех байт (или из шестнадцати — в версии протокола IPv6), таких как 132.163.4.102. Тем не менее, вы можете использовать класс `InetAddress`, если необходимо выполнить преобразование между именами хостов и Internet-адресами. Пакет `java.net` поддерживает Internet-адреса в формате протокола IPv6, при условии, что и операционная система хоста тоже их поддерживает.

Статический метод `getByName` возвращает объект `InetAddress` хоста. Например,

```
InetAddress address = InetAddress.getByName("time-A.timefreq.bldrdoc.gov");
```

возвращает объект `InetAddress`, инкапсулирующий последовательность из четырех байт 132.163.4.104. Вы можете получить доступ к этим байтам с помощью метода `getAddress`:

```
byte[] addressBytes = address.getAddress();
```

Имена некоторых хостов, обслуживающих большой объем трафика, соответствуют нескольким Internet-адресам, что объясняется попыткой сбалансировать нагрузку. Например, на момент написания этой книги имя хоста `java.sun.com` соответствовало трем различным Internet-адресам. Один из них выбирается случайным образом во время доступа к хосту. Получить все хосты можно посредством метода `getAllByName`:

```
InetAddress[] addresses = InetAddress.getAllByName(host);
```

И, наконец, в некоторых случаях вам будет необходим адрес локального хоста. Если вы запросите адрес `localhost`, то неизменно получите адрес 127.0.0.1, который другие не могут использовать для подключения к вашему компьютеру. Вместо этого нужно применить метод `getLocalHost`, чтобы получить адрес вашего локального хоста:

```
InetAddress address = InetAddress.getLocalHost();
```

В листинге 3.2 приведен пример простой программы, выводящей на экран Internet-адрес вашего локального хоста, если не будут определены какие-либо параметры командной строки, или все Internet-адреса другого хоста, если вы определите имя хоста в командной строке, например:

```
java InetAddressTest java.sun.com
```

Листинг 3.2. Содержимое файла `InetAddressTest.java`

```
1.  import java.net.*;
2.
3.  /**
4.   * Данная программа демонстрирует пример использования класса InetAddress.
5.   * Вам необходимо задать имя хоста в качестве аргумента командной строки
6.   * или запустить программу без аргументов командной строки, чтобы
7.   * посмотреть адрес локального хоста.
8.   * @version 1.01 2001-06-26
9.   * @author Cay Horstmann
10.  */
11. public class InetAddressTest
12. {
13.    public static void main(String[] args)
14.    {
15.       try
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.       catch (Exception e)
31.       {
32.          e.printStackTrace();
33.       }
34.    }
35. }
```

API `java.net.InetAddress 1.0`

- `static InetAddress getByName(String host)`
- `static InetAddress[] getAllByName(String host)`
  Создает `InetAddress`, или массив всех Internet-адресов, для данного имени хоста.
- `static InetAddress getLocalHost()`
  Создает `InetAddress` для локального хоста.
- `byte[] getAddress()`
  Возвращает массив байтов, содержащий числовой адрес.
- `String getHostAddress()`
  Возвращает строку с десятичными числами, например, "`132.163.4.102`".
- `String getHostName()`
  Возвращает имя хоста.

Реализация серверов

Теперь после создания клиента, который может получать информацию из сети, попробуем создать простой сервер, который может посылать данные. После запуска программа-сервер переходит в режим ожидания обращения клиентов по своему порту. Выберите номер порта 8189, который не используется никакими стандартными устройствами. Приведенная ниже команда позволяет создать сервер с портом 8189:

```
ServerSocket s = new ServerSocket(8189);
```

Следующая команда сообщает программе, что она должна ожидать обращения клиентов по заданному порту:

```
Socket incoming = s.accept();
```

Сразу после обращения клиента (посредством переданного по сети запроса) этот метод возвращает объект `Socket`, который представляет установленное соединение. Данный объект можно использовать для чтения входных и записи выходных данных так, как показано ниже.

```
InputStream inStream = incoming.getInputStream();
OutputStream outStream = incoming.getOutputStream();
```

Все данные выходного потока сервера становятся данными входного потока клиента и наоборот: все данные выходного потока клиента поступают во входной поток сервера. Во всех примерах этой главы текстовые данные в потоках будут передаваться через сокеты. Потоки, связанные с сокетом, будут преобразовываться в объекты `Scanner` и `Writer`.

```
Scanner in = new Scanner(inStream);
PrintWriter out = new PrintWriter(outStream, true
   /* автоматическая передача оставшихся данных */);
```

Предположим, что программа-клиент отправила приветствие:

```
out.println("Hello! Enter BYE to exit.");
```

При использовании утилиты `telnet` для подключения к программе-серверу через порт 8189 данное приветствие будет отображено в окне терминала.

В этой простой программе-сервере построчно считываются входные данные, отправленные программой-клиентом, и отображаются на экране в режиме “эхо”. Подобным образом пример демонстрирует получение входных данных от программы-клиента. Программа-сервер должна обработать полученные данные и дать адекватный ответ.

```
String line = in.nextLine();
out.println("Echo: " + line);
if (line.trim().equals("BYE")) done = true;
```

В конце работы следует закрыть используемый сокет:

```
incoming.close();
```

Вот и все. Любая программа-сервер, например, Web-сервер, выполняет такой же цикл основных действий.

1. Получение команды от программы-клиента (“дайте мне нужную информацию”) из входного потока.
2. Расшифровка команды клиента.
3. Сбор информации, запрошенной клиентом.
4. Передача клиенту найденной информации через выходной поток.

В листинге 3.3 показан полный текст описанной выше программы-сервера.

Листинг 3.3. Содержимое файла `EchoServer.java`

```
1.  import java.io.*;
2.  import java.net.*;
3.  import java.util.*;
4.
5.  /**
6.   * Данная программа реализует простой сервер, который ожидает
7.   * обращения по порту 8189 и возвращает клиенту данные,
8.   * полученные от него.
9.   * @version 1.20 2004-08-03
10.  * @author Cay Horstmann
11.  */
12. public class EchoServer
13. {
14.    public static void main(String[] args )
15.    {
16.       try
17.       {
18.          // Создание сокета на стороне сервера.
19.          ServerSocket s = new ServerSocket(8189);
20.
21.          // Ожидание обращения клиента.
22.          Socket incoming = s.accept( );
23.          try
24.          {
25.             InputStream inStream = incoming.getInputStream();
26.             OutputStream outStream = incoming.getOutputStream();
27.
28.             Scanner in = new Scanner(inStream);
29.             PrintWriter out = new PrintWriter(outStream, true
30.                /* автоматическая передача оставшихся данных */);
31.
32.             out.println( "Hello! Enter BYE to exit." );
33.
34.             // Передача клиенту полученных от него данных.
35.             boolean done = false;
36.             while (!done && in.hasNextLine())
37.             {
38.                String line = in.nextLine();
39.                out.println("Echo: " + line);
40.                if (line.trim().equals("BYE")) done = true;
41.             }
42.          }
43.          finally
44.          {
45.             incoming.close();
46.          }
47.       }
48.       catch (IOException e)
49.       {
50.          e.printStackTrace();
51.       }
52.    }
53. }
```

Для проверки работоспособности программы ее нужно откомпилировать и запустить. Затем необходимо подключиться с помощью утилиты `telnet` к серверу `localhost` (или к IP-адресу 127.0.0.1) и по порту 8189.

Если ваш компьютер непосредственно подключен к Internet, любой пользователь может получить доступ к программе-серверу при условии, что ему известен IP-адрес и номер порта.

При обращении по этому порту будет получено сообщение, показанное на рис. 3.4:

```
Hello! Enter BYE to exit.
```

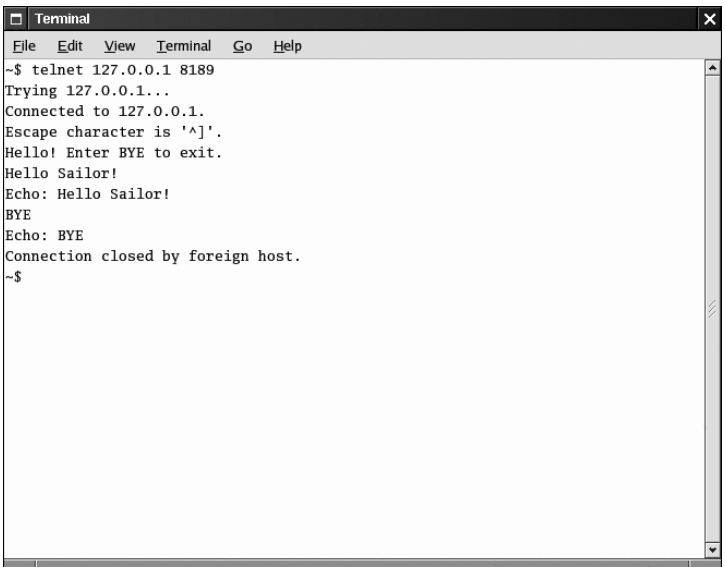


Рис. 3.4. Пример работы сервера, который передает клиенту полученные от него данные

Введите любую фразу, и вы получите ее в неизменном виде. Для отключения от программы-сервера введите `BYE` (все символы в верхнем регистре).

API java.net.ServerSocket 1.0

- `ServerSocket(int port)`

  Создает сокет на стороне сервера, который осуществляет мониторинг порта.

- `Socket accept()`

  Ожидает соединения. Этот метод блокирует (т.е. переводит в режим ожидания) текущий поток до тех пор, пока не будет совершено подключение. Метод возвращает объект `Socket`, через который программа может взаимодействовать с подключенным клиентом.

- `void close()`

  Закрывает сокет на стороне сервера.

Обслуживание множества серверов

В предыдущем простом примере программы-сервера не предусмотрена возможность одновременного подключения сразу нескольких программ-клиентов. Обычно

программа-сервер работает на компьютере-сервере, а программы-клиенты могут одновременно подключаться к ней по Internet из любой точки мира. Если на сервере не предусмотрена обработка одновременного обращения нескольких клиентов, то это приведет к тому, что один клиент может монополизировать доступ к данной программе в течение длительного времени. Во избежание таких ситуаций следует прибегнуть к помощи потоков.

Для каждого нового соединения, т.е. при успешной обработке запроса на соединение, будет запущен новый поток, который позаботится об организации взаимодействия между программой-сервером и *данной* программой-клиентом. Для этого в программе-сервере следует организовать показанный ниже 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 программой-клиентом.

```
class ThreadedEchoHandler implements Runnable
{  . . .
   public void run()
   {
      try
      {
         InputStream inStream = incoming.getInputStream();
         OutputStream outStream = incoming.getOutputStream();
         process input and send response
         incoming.close();
      }
      catch(IOException e)
      {
         обработка исключения
      }
   }
}
```

Теперь несколько программ-клиентов могут одновременно подключаться к серверу, поскольку для каждого соединения создается новый поток. Это можно легко проверить.

1. Скомпилируйте и запустите серверную программу, код которой приведен в листинге 3.4.
2. Откройте несколько окон утилиты `telnet` (рис. 3.5).
3. Переключайтесь между окнами и вводите команды. Теперь каждое отдельное окно утилиты `telnet` может независимо взаимодействовать с программой-сервером.
4. Для того чтобы разорвать соединение и закрыть окно утилиты `telnet`, нажмите комбинацию клавиш <Ctrl+C>.

В данной программе мы порождаем отдельный поток для каждого соединения. Этот подход не вполне приемлем для высокопроизводительного сервера. Более эффективной работы сервера можно добиться, используя средства из пакета `java.nio`. Дополнительную информацию по этому вопросу можно получить, обратившись по адресу `http://www-106.ibm.com/developerworks/java/library/j-javaio`.

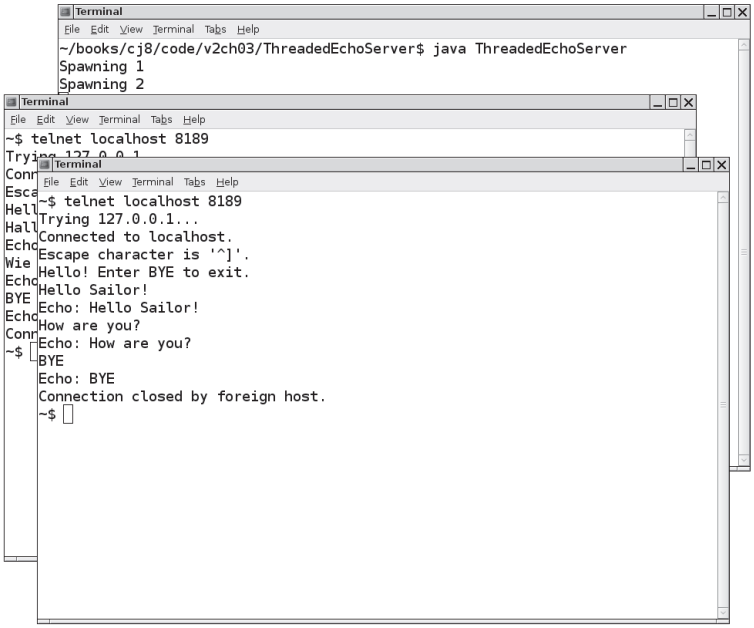


Рис. 3.5. Пример одновременного доступа к многопоточной программе-серверу

Листинг 3.4. Содержимое файла `ThreadedEchoServer.java`

```
1.  import java.io.*;
2.  import java.net.*;
3.  import java.util.*;
4.
5.  /**
6.   * Данная программа реализует многопоточный сервер, который
7.   * ожидает обращения по порту 8189 и возвращает клиенту
8.   * переданные им данные.
9.   * @author Cay Horstmann
10.  * @version 1.20 2004-08-03
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)
         {
            Socket incoming = s.accept();
            System.out.println("Spawning " + i);
            Runnable r = new ThreadedEchoHandler(incoming, i);
            Thread t = new Thread(r);
            t.start();
            i++;
         }
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
   }
}

/**
   Этот класс обрабатывает данные введенные клиентом
   в рамках одного соединения.
*/
class ThreadedEchoHandler implements Runnable
{
   /**
      Конструктор обработчика.
      @param i Сокет для соединения
      @param c Счетчик обработчиков
   */
   public ThreadedEchoHandler(Socket i)
   {
      incoming = i;
   }
   public void run()
   {
      try
      {
         try
         {
            InputStream inStream = incoming.getInputStream();
            OutputStream outStream = incoming.getOutputStream();

            Scanner in = new Scanner(inStream);
            PrintWriter out = new PrintWriter(outStream, true
               /* автоматическая передача оставшихся данных */);

            out.println( "Hello! Enter BYE to exit." );

            // Передача клиенту полученных от него данных
            boolean done = false;
            while (!done && in.hasNextLine())
            {
               String line = in.nextLine();
               out.println("Echo: " + line);
               if (line.trim().equals("BYE"))
                  done = true;
            }
         }
```

```
78.          finally
79.          {
80.             incoming.close();
81.          }
82.       }
83.       catch (IOException e)
84.       {
85.          e.printStackTrace();
86.       }
87.    }
88.
89.    private Socket incoming;
90. }
```

Одностороннее закрытие

Одностороннее закрытие (half-close) обеспечивает возможность одной стороне закрыть соединение с сокетом, прекратив отправку своих данных, но при этом получая данные с другой стороны.

Вот типичная ситуация. Предположим, что вы отправляете данные на сервер, но не знаете, какой объем данных необходимо передать. Если вы имеете дело с файлом, то, закрыв его, вы тем самым определяете конец данных. Закрыв же сокет, вы немедленно разорвете соединение с сервером.

Данную проблему решает одностороннее закрытие. Если вы закроете выходной поток, связанный с сокетом, вы тем самым укажете серверу на окончание запроса. При этом входной поток останется открытым, и вы сможете прочитать ответ.

Программа на стороне клиента, реализующая данный подход, выглядит приблизительно так, как показано ниже.

```
Socket socket = new Socket(host, port);
Scanner in = new Scanner(socket.getInputStream());
PrintWriter writer = new PrintWriter(socket.getOutputStream());
// Передача данных запроса.
writer.print(. . .);
writer.flush();
socket.shutdownOutput();
// Теперь один из потоков, связанных с сокетом, закрыт.
// Чтение ответа.
while (in.hasNextLine()) != null)
   { String line = in.nextLine(); . . . }
socket.close();
```

Программа на стороне сервера лишь читает данные из входного потока до тех пор, пока не закроется выходной поток на другом конце соединения.

Очевидно, что данный подход применим только при работе с протоколами, подобными HTTP, где клиент устанавливает соединение с сервером, передает запрос, получает ответ, после чего соединение разрывается.

API java.net.Socket 1.0

- `void shutdownOutput()` 1.3
  Определяет “конец потока” для выходного потока.
- `void shutdownInput()` 1.3
  Определяет “конец потока” для входного потока.

- `boolean isOutputShutdown()` 1.4
  Возвращает значение `true`, если вывод данных был остановлен.
- `boolean isInputShutdown()` 1.4
  Возвращает значение `true`, если вывод данных был остановлен.

Прерываемые сокеты

При подключении с помощью сокета текущий поток блокируется до тех пор, пока соединение не будет установлено, или до истечения времени тайм-аута. Аналогично, если вы пытаетесь передать или прочитать данные посредством сокета, поток приостановит выполнение до успешного завершения операции или до окончания времени тайм-аута.

В реальных приложениях желательно предоставить пользователям возможность прервать слишком затянувшийся процесс установления соединения с помощью сокета. Однако если поток блокирован, поскольку не получает ответа от сокета, вы не можете разблокировать его, вызвав метод `interrupt()`.

Для прерывания операций с сокетом используется класс `SocketChannel`, предоставляемый пакетом `java.nio`. Объект `SocketChannel` создается следующим образом:

```
SocketChannel channel = SocketChannel.open(new
   InetSocketAddress(host, port));
```

С каналом не связываются потоки. Вместо этого он предоставляет методы `read()` и `write()`, использующие объекты `Buffer`. (Дополнительную информацию о буферах NIO можно найти в главе 1.) Данные методы объявлены в интерфейсах `ReadableByteChannel` и `WritableByteChannel`.

Если вы не хотите работать с буферами, можете использовать для чтения из `SocketChannel` объект `Scanner`. В классе `Scanner` предусмотрен конструктор с параметром `ReadableByteChannel`:

```
Scanner in = new Scanner(channel);
```

Для формирования выходного потока на базе канала применяется статический метод `Channels.newOutputStream()`:

```
OutputStream outStream = Channels.newOutputStream(channel);
```

Вот и все, что необходимо сделать. Если поток будет прерван в процессе открытия соединения, чтения или записи, соответствующая операция завершится и сгенерирует исключение.

Программа, код которой показан в листинге 3.5, демонстрирует прерываемые и блокирующие сокеты. Сервер передает числа, и должен остановить передачу после десятого числа. Щелкните на любой кнопке, чтобы запустить поток, устанавливающий соединение с сервером и выводящий выходные данные. Первый поток использует прерываемый сокет, а второй — блокирующий сокет. Если щелкнуть на кнопке Cancel (Отмена) во время вывода первых десяти чисел, вы прервете любой поток.

А если вы щелкнете на кнопке Cancel после первых десяти чисел, то вы прервете только первый поток. Второй поток будет продолжать блокирование до тех пор, пока сервер, в конце концов, не завершит соединение (рис. 3.6).



Рис. 3.6. Прерывание операций с сокетом

Листинг 3.5. Содержимое файла `InterruptibleSocketTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.util.*;
4.  import java.net.*;
5.  import java.io.*;
6.  import java.nio.channels.*;
7.  import javax.swing.*;
8.
9.  /**
10.  * Данная программа демонстрирует возможность прерывания
11.  * при использовании канала сокета.
12.  * @author Cay Horstmann
13.  * @version 1.01 2007-06-25
14.  */
15. public class InterruptibleSocketTest
16. {
17.    public static void main(String[] args)
18.    {
19.       EventQueue.invokeLater(new Runnable()
20.          {
21.             public void run()
22.
23.             {
24.                JFrame frame = new InterruptibleSocketFrame();
25.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26.                frame.setVisible(true);
27.             }
28.          });
29.    }
30. }
31.
32. class InterruptibleSocketFrame extends JFrame
33. {
34.    public InterruptibleSocketFrame()
35.    {
36.       setSize(WIDTH, HEIGHT);
37.       setTitle("InterruptibleSocketTest");
38.
```

```
39.       JPanel northPanel = new JPanel();
40.       add(northPanel, BorderLayout.NORTH);
41.
42.       messages = new JTextArea();
43.       add(new JScrollPane(messages));
44.
45.       interruptibleButton = new JButton("Interruptible");
46.       blockingButton = new JButton("Blocking");
47.
48.       northPanel.add(interruptibleButton);
49.       northPanel.add(blockingButton);
50.
51.       interruptibleButton.addActionListener(new ActionListener()
52.          {
53.             public void actionPerformed(ActionEvent event)
54.             {
55.                interruptibleButton.setEnabled(false);
56.                blockingButton.setEnabled(false);
57.                cancelButton.setEnabled(true);
58.                connectThread = new Thread(new Runnable()
59.                   {
60.                      public void run()
61.                      {
62.                         try
63.                         {
64.                            connectInterruptibly();
65.                         }
66.                         catch (IOException e)
67.                         {
68.                            messages.append(
69.                            "\nInterruptibleSocketTest.connectInterruptibly: " + e);
70.                         }
71.                      }
72.                   });
73.                connectThread.start();
74.             }
75.          });
76.
77.       blockingButton.addActionListener(new ActionListener()
78.          {
79.             public void actionPerformed(ActionEvent event)
80.             {
81.                interruptibleButton.setEnabled(false);
82.                blockingButton.setEnabled(false);
83.                cancelButton.setEnabled(true);
84.                connectThread = new Thread(new Runnable()
85.                   {
86.                      public void run()
87.                      {
88.                         try
89.                         {
90.                            connectBlocking();
91.                         }
92.                         catch (IOException e)
93.                         {
94.                            messages.append(
95.                              "\nInterruptibleSocketTest.connectBlocking: " + e);
```

```
96.                      }
97.                   }
98.                });
99.             connectThread.start();
100.         }
101.      });
102.
103.   cancelButton = new JButton("Cancel");
104.   cancelButton.setEnabled(false);
105.   northPanel.add(cancelButton);
106.   cancelButton.addActionListener(new ActionListener()
107.      {
108.         public void actionPerformed(ActionEvent event)
109.         {
110.            connectThread.interrupt();
111.            cancelButton.setEnabled(false);
112.         }
113.      });
114.   server = new TestServer();
115.   new Thread(server).start();
116. }
117. /**
118.  * Соединение с тестовым сервером посредством прерываемого ввода-вывода.
119.  */
120. public void connectInterruptibly() throws IOException
121. {
122.   messages.append("Interruptible:\n");
123.   SocketChannel channel =
124.      SocketChannel.open(new InetSocketAddress("localhost", 8189));
125.   try
126.   {
127.      in = new Scanner(channel);
128.      while (!Thread.currentThread().isInterrupted())
129.      {
130.         messages.append("Reading ");
131.         if (in.hasNextLine())
132.         {
133.            String line = in.nextLine();
134.            messages.append(line);
135.            messages.append("\n");
136.         }
137.      }
138.   }
139.   finally
140.   {
141.      channel.close();
142.      EventQueue.invokeLater(new Runnable()
143.      {
144.         public void run()
145.         {
146.            messages.append("Channel closed\n");
147.            interruptibleButton.setEnabled(true);
148.            blockingButton.setEnabled(true);
149.         }
150.      });
151.   }
152. }
153.
```

```
154.    /**
155.     * Соединение с тестовым сервером посредством блокирующего ввода-вывода.
156.     */
157.    public void connectBlocking() throws IOException
158.    {
159.       messages.append("Blocking:\n");
160.       Socket sock = new Socket("localhost", 8189);
161.       try
162.       {
163.          in = new Scanner(sock.getInputStream());
164.          while (!Thread.currentThread().isInterrupted())
165.          {
166.             messages.append("Reading ");
167.             if (in.hasNextLine())
168.             {
169.                String line = in.nextLine();
170.                messages.append(line);
171.                messages.append("\n");
172.             }
173.          }
174.       }
175.       finally
176.       {
177.          sock.close();
178.          EventQueue.invokeLater(new Runnable()
179.          {
180.             public void run()
181.             {
182.                messages.append("Socket closed\n");
183.                interruptibleButton.setEnabled(true);
184.                blockingButton.setEnabled(true);
185.             }
186.          });
187.       }
188.    }
189.
190.    /**
191.     * Многопоточный сервер, который ожидает обращения
192.     * по порту 8189 и передает клиенту псевдослучайные значения,
193.     * имитируя останов сервера через 10 чисел.
194.     */
195.    class TestServer implements Runnable
196.    {
197.       public void run()
198.       {
199.          try
200.          {
201.             ServerSocket s = new ServerSocket(8189);
202.             while (true)
203.             {
204.                Socket incoming = s.accept();
205.                Runnable r = new TestServerHandler(incoming);
206.                Thread t = new Thread(r);
207.                t.start();
208.             }
209.          }
```

```
210.         catch (IOException e)
211.         {
212.            messages.append("\nTestServer.run: " + e);
213.         }
214.      }
215.   }
216.
217.   /**
218.    * Данный класс обрабатывает данные, полученные
219.    * от клиента в рамках одного соединения.
220.    */
221.   class TestServerHandler implements Runnable
222.   {
223.      /**
224.       * Конструктор обработчика.
225.       * @param i — входящий сокет
226.       */
227.      public TestServerHandler(Socket i)
228.      {
229.         incoming = i;
230.      }
231.
232.      public void run()
233.      {
234.         try
235.         {
236.            OutputStream outStream = incoming.getOutputStream();
237.            PrintWriter out = new PrintWriter(outStream, true /* autoFlush */);
238.            while (counter < 100)
239.            {
240.               counter++;
241.               if (counter <= 10) out.println(counter);
242.               Thread.sleep(100);
243.            }
244.            incoming.close();
245.            messages.append("Closing server\n");
246.         }
247.         catch (Exception e)
248.         {
249.            messages.append("\nTestServerHandler.run: " + e);
250.         }
251.      }
252.      private Socket incoming;
253.      private int counter;
254.   }
255.
256.   private Scanner in;
257.   private JButton interruptibleButton;
258.   private JButton blockingButton;
259.   private JButton cancelButton;
260.   private JTextArea messages;
261.   private TestServer server;
262.   private Thread connectThread;
263.
264.   public static final int WIDTH = 300;
265.   public static final int HEIGHT = 300;
266. }
```

API java.net.InetSocketAddress 1.4

- `InetSocketAddress(String hostname, int port)`
  Создает объект адреса для указанного узла и порта. Преобразование имени узла осуществляется в процессе установления соединения. Если преобразовать имя в адрес не удалось, устанавливается значения `true` свойства `unresolved`.
- `boolean isUnresolved()`
  Возвращает `true`, если преобразование имени в адрес выполнить не удалось.

API java.nio.channels.SocketChannel 1.4

- `static SocketChannel open(SocketAddress address)`
  Открывает канал сокета и связывает его с удаленным узлом.

API java.nio.channels.Channels 1.4

- `static InputStream newInputStream(ReadableByteChannel channel)`
  Создает входной поток, поддерживающий чтение из указанного канала.
- `static OutputStream newOutputStream(WritableByteChannel channel)`
  Создает выходной поток, поддерживающий запись в указанный канал.

Отправка электронной почты

В этом разделе приведен пример практического использования программирования сокетов для отправки электронной почты удаленному компьютеру.

Для отправки электронной почты необходимо установить соединение с сокетом по порту 25, который обычно используется для протокола SMTP (Simple Mail Transport Protocol — простой протокол передачи почты). Протокол SMTP описывает формат электронных сообщений. Вы можете подключаться к любому серверу, на котором выполняется служба SMTP. Однако сервер должен быть готов к приему запроса на соединение. Ранее серверы с демонами `sendmail` могли принимать любые электронные сообщения, но в настоящее время из-за большого объема спама (т.е. массовой рассылки электронных сообщений с навязчивой рекламой и другим бесполезным содержанием) в большинстве серверов встроена проверка и допуск почтовых сообщений только от разрешенных пользователей или IP-адресов.

Сразу после соединения с сервером следует послать заголовок сообщения (в формате SMTP, который достаточно просто создать), а затем и текст сообщения так, как показано ниже.

1. Создайте сокет на вашем компьютере:

```
Socket s = new Socket("mail.yourserver.com", 25); // порт 25 соответствует SMTP
PrintWriter out = new PrintWriter(s.getOutputStream());
```

2. В выходном потоке передайте следующую информацию:

```
HELO компьютер-отправитель
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 заканчивались символами /r и /n.

На некоторых SMTP-серверах достоверность данных не проверяется, т.е. можно передать любую информацию об отправителе. (Имейте в виду, что любой желающий может послать поддельное сообщение, например, с обратным адресом president@whitehouse.gov и приглашением посетить прием на лужайке Белого дома.)

В листинге 3.6 приведен очень простой пример почтовой программы. Как показано на рис. 3.7, с ее помощью можно ввести адрес отправителя, адрес получателя, имя SMTP-сервера, текст сообщения, а затем отправить данное сообщение с помощью кнопки Send (Отправить).

Эта программа просто передает SMTP-серверу последовательность команд, которая обсуждалась ранее. На экране эти команды отображаются вместе с ответами.

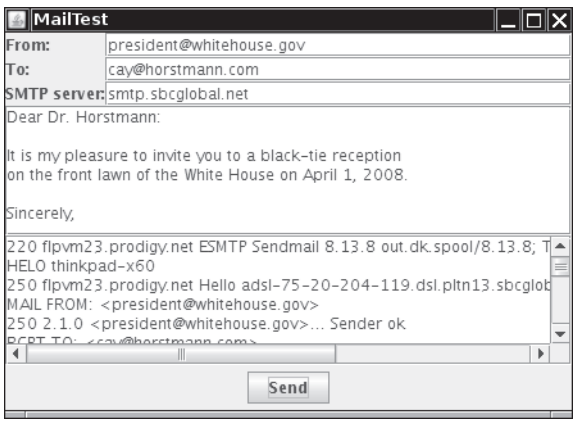


Рис. 3.7. Программа `MailTest`



Когда эта программа появилась в первом издании настоящей книги в 1996 г., большинство SMTP-серверов принимали соединения, которые поступали откуда угодно, вовсе не проверяя их. В наше время большинство серверов не являются столь доверчивыми, поэтому с запуском этой программы у вас могут возникнуть трудности. Почтовый сервер вашего поставщика услуг Internet может принимать соединения, поступающие из вашего дома или из доверенного IP-адреса. Другие серверы используют правило “POP before SMTP”, требующее, чтобы вы сначала загрузили свою электронную почту (для этого потребуется ввести пароль), прежде чем отправлять какие-либо сообщения. Попробуйте ввести ложный адрес электронной почты, прежде чем отправлять письмо с помощью этой программы. Все большую популярность приобретает расширение SMTP, требующее применения зашифрованного пароля (http://tools.ietf.org/html/rfc2554). Наша простая программа не поддерживает упомянутый механизм аутентификации.

В последнем разделе вы могли видеть, как используются приемы программирования на уровне сокетов для подключения к SMTP-серверу и отправки электронного сообщения. Хорошей новостью является то, что мы это можем сделать; более того, с помощью нашей программы вы сможете понять суть такой важной службы Internet, как электронная почта. Тем не менее, если вы планируете разработать приложение, которое будет включать систему обработки электронной почты, вам придется работать на более высоком уровне и использовать библиотеку, инкапсулирующую подробности протоколов. Например, компания Sun Microsystems разработала JavaMail API в

качестве стандартного расширения платформы Java. Чтобы отправить сообщение в JavaMail API, нужно просто сделать следующий вызов:

```
Transport.send(message);
```

Библиотека позаботится о протоколах сообщений, аутентификации, обработке прикрепленных файлов и т.п.

Листинг 3.6. Содержимое файла `MailTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.util.*;
4.  import java.net.*;
5.  import java.io.*;
6.  import javax.swing.*;
7.
8.  /**
9.   * Данная программа демонстрирует использование
10.  * сокетов для передачи почтовых сообщений.
11.  * @author Cay Horstmann
12.  * @version 1.11 2007-06-25
13.  */
14. public class MailTest
15. {
16.   public static void main(String[] args)
17.   {
18.      EventQueue.invokeLater(new Runnable)
19.         {
20.            public void run()
21.            {
22.               JFrame frame = new MailTestFrame();
23.               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24.               frame.setVisible(true);
25.            }
26.         });
27.   }
28. }
29.
30. /**
31.  * Фрейм для пользовательского интерфейса.
32.  */
33. class MailTestFrame extends JFrame
34. {
35.   public MailTestFrame()
36.   {
37.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
38.      setTitle("MailTest");
39.
40.      setLayout(new GridBagLayout());
41.
42.      // Мы используем класс GBC, описанный в главе 9 первого тома.
43.      add(new JLabel("From:"), new GBC(0,
44.         0).setFill(GBC.HORIZONTAL));
45.
46.      from = new JTextField(20);
47.      add(from, new GBC(1,
48.         0).setFill(GBC.HORIZONTAL).setWeight(100, 0));
```

```
49.       add(new JLabel("To:"), new GBC(0,
50.         1).setFill(GBC.HORIZONTAL));
51.
52.       to = new JTextField(20);
53.       add(to, new GBC(1, 1).setFill(GBC.HORIZONTAL).
54.          setWeight(100, 0));
55.
56.       add(new JLabel("SMTP server:"), new GBC(0,
57.         2).setFill(GBC.HORIZONTAL));
58.
59.       smtpServer = new JTextField(20);
60.       add(smtpServer, new GBC(1,
61.         2).setFill(GBC.HORIZONTAL).setWeight(100, 0));
62.
63.       message = new JTextArea();
64.       add(new JScrollPane(message), new GBC(0,
65.         3, 2, 1).setFill(GBC.BOTH).setWeight(100, 100));
66.
67.       comm = new JTextArea();
68.       add(new JScrollPane(comm), new GBC(0,
69.         4, 2, 1).setFill(GBC.BOTH).setWeight(100, 100));
70.
71.       JPanel buttonPanel = new JPanel();
72.       add(buttonPanel, new GBC(0, 5, 2, 1));
73.
74.       JButton sendButton = new JButton("Send");
75.       buttonPanel.add(sendButton);
76.       sendButton.addActionListener(new ActionListener()
77.         {
78.           public void actionPerformed(ActionEvent event)
79.           {
80.             new SwingWorker<Void, Void>()
81.             {
82.               protected Void doInBackground() thrown Exception
83.                  {
84.                  comm.setText("");
85.                  sendMail();
86.                  return null;
87.               }
88.             }.execute();
89.           }
90.         });
91.     }
92.     /**
93.      * Передача почтового сообщения, которое было задано
94.      * с применением средств пользовательского интерфейса.
95.      */
96.     public void sendMail()
97.     {
98.       try
99.       {
100.        Socket s = new Socket(smtpServer.getText(), 25);
101.
102.        InputStream inStream = s.getInputStream();
103.        OutputStream outStream = s.getOutputStream();
104.
105.        in = new Scanner(inStream);
106.        out = new PrintWriter(outStream, true /* autoFlush */);
```

```
107.        String hostName = InetAddress.getLocalHost().getHostName();
108.
109.        receive();
110.        send("HELO " + hostName);
111.        receive();
112.        send("MAIL FROM: <" + from.getText() + ">");
113.        receive();
114.        send("RCPT TO: <" + to.getText() + ">");
115.        receive();
116.        send("DATA");
117.        receive();
118.        send(message.getText());
119.        send(".");
120.        receive();
121.        s.close();
122.     }
123.     catch (IOException e)
124.     {
125.        comm.append("Error: " + e);
126.     }
127.  }
128.  /**
129.   * Передача строки сокету и воспроизведение ее
130.   * в текстовой области comm.
131.   * @param s Строка для передачи.
132.   */
133.  public void send(String s) throws IOException
134.  {
135.     comm.append(s);
136.     comm.append("\n");
137.     out.print(s.replaceAll("\n", "\r\n"));
138.     out.print("\r\n");
139.     out.flush();
140.  }
141.  /**
142.     Получение строки посредством сокета и отображение
143.     ее в текстовой области comm.
144.  */
145.  public void receive() throws IOException
146.  {
147.     String line = in.nextLine();
148.     comm.append(line);
149.     comm.append("\n");
150.  }
151.
152.  private Scanner in;
153.  private PrintWriter out;
154.  private JTextField from;
155.  private JTextField to;
156.  private JTextField smtpServer;
157.  private JTextArea message;
158.  private JTextArea comm;
159.
160.  public static final int DEFAULT_WIDTH = 300;
161.  public static final int DEFAULT_HEIGHT = 300;
162. }
```

Создание соединений с URL

Чтобы получить доступ к Web-серверам в программе на Java, вам придется работать на более высоком уровне, чем создание подключения к сокету и выдача HTTP-запросов. В следующих разделах мы поговорим о классах, которые библиотека Java предлагает для этой цели.

URL и URI

Классы `URL` и `URLConnection` инкапсулируют большую часть всех сложных подробностей процесса извлечения информации с удаленного узла. Объект `URL` создается следующим образом:

```
URL url = new URL(строка_URL);
```

Для простого извлечения содержимого заданного ресурса достаточно использовать метод `openStream()` класса `URL`. Этот метод возвращает объект `InputStream`. Поток данного типа можно использовать обычным образом, например? создать объект `Scanner`:

```
InputStream inStream = url.openStream();
Scanner in = new Scanner(inStream);
```

Пакет `java.net` различает унифицированные *указатели* ресурсов (uniform resource locator — URL) и унифицированные *идентификаторы* ресурсов (uniform resource identificator — URI).

URI — это лишь синтаксическая конструкция, которая содержит различные части строки, описывающей Web-ресурс. URL — это особый вид URI-идентификатора с информацией о расположении ресурса. Например, URI `mailto:cay@hortsmann.com` не является *указателем*, потому что не указывает на какие-либо данные. Такой URI называется *унифицированным именем ресурса* (uniform resource name — URN).

В библиотеке Java класс `URI` не имеет методов доступа к указанному им ресурсу, так как предназначен только для синтаксического разбора строки. В отличие от него, класс `URL` может открыть поток для данного ресурса. Поэтому класс `URL` может работать только со схемами, которые библиотека Java "умеет" обрабатывать, в частности, `http:`, `https:`, `ftp:`, указателями на позиции локальной файловой системы (`file:`) и JAR-файлы (`jar:`).

Разбор URI — задача далеко не тривиальная, поскольку идентификаторы ресурсов могут иметь сложную структуру. В качестве примера рассмотрим следующие URI:

```
http://maps.yahoo.com/py/maps.py?csz=Cupertino+CA
ftp://username:password@ftp.yourserver.com/pub/file.txt
```

Спецификация URI задает правила построения этих идентификаторов. Структура URI выглядит следующим образом:

```
[схема:]специальная_часть[#фрагмент]
```

Здесь квадратные скобки (`[]`) обозначают необязательную часть, а двоеточие и символ # являются разделителями. Если *`схема:`* присутствует в URI, то он называется *абсолютным*, а в противном случае — *относительным*.

Абсолютный URI называется *непрозрачным*, если *`специальная_часть`* не начинается с косой черты (`/`), например, имеет вид

```
mailto:cay@hortsmann.com
```

Все абсолютные прозрачные URI и все относительные URL имеют *иерархическую* структуру. Соответствующие примеры приведены ниже:

```
http://java.sun.com/index.html
../../java/net/Socket.html#Socket()
```

Компонент *специальная_часть* иерархического URI имеет такую структуру:

```
[/полномочия][путь][?запрос]
```

Здесь квадратные скобки ([]) тоже обозначают необязательную часть.

В URI серверов компонент *полномочия* имеет следующую форму:

```
[информация_о_пользователе@]имя_компьютера[:порт]
```

Элемент *порт* должен иметь целочисленное значение.

В документе RFC 2396, определяющем стандарт URI, также допускается механизм указания этой части в другом формате на основе информации из реестра, но он не получил широкого распространения.

Одна из основных задач класса `URI` заключается в синтаксическом разборе отдельных компонентов идентификатора и извлечении их с помощью перечисленных ниже методов:

```
getScheme
getSchemeSpecificPart
getAuthority
getUserInfo
getHost
getPort
getPath
getQuery
getFragment
```

Другая задача класса `URI` состоит в обработке абсолютных и относительных идентификаторов. Предположим, например, что в нашем распоряжении есть абсолютный и относительный URI:

```
http://docs.company.com/api/java/net/ServerSocket.html
../../java/net/Socket.html#Socket()
```

На основании этой информации можно определить абсолютный URI, соответствующий относительному идентификатору:

```
http://docs.company.com/api/java/net/Socket.html#Socket()
```

Этот процесс называется *преобразованием* (resolving) относительного URI.

Обратный процесс называется *преобразованием абсолютных адресов в относительные* (relativization). Например, имея *базовый* URI `http://docs.mycompany.com/api`, можно получить для абсолютного URI `http://docs.company.com/api/java/lang/String.html` относительный URI `java/lang/String.html`.

Для выполнения этих операций в классе URI предусмотрено два соответствующих метода:

```
relative = base.relativize(combined);
combined = base.resolve(relative);
```

Извлечение информации с помощью `URLConnection`

Для получения дополнительной информации о ресурсе потребуется использовать класс `URLConnection`, который предоставляет гораздо больше средств управления доступом к Web-ресурсам.

При работе с объектом `URLConnection` необходимо тщательно продумать и выполнить все перечисленные ниже действия.

1. Вызвать метод `openConnection()` класса `URL` для получения объекта `URLConnection`.

   ```
   URLConnection connection = url.openConnection();
   ```

2. Указать свойства запроса с помощью перечисленных ниже методов:

   ```
   setDoInput
   setDoOutput
   setIfModifiedSince
   setUseCaches
   setAllowUserInteraction
   setRequestProperty
   ```

   Эти методы будут подробно рассматриваться далее.

3. Установить соединение с удаленным ресурсом с помощью метода `connect()`:

   ```
   connection.connect();
   ```

   Помимо создания сокета, для установления соединения с сервером этот метод также запрашивает у сервера *заголовок*.

4. После подключения к серверу становятся доступны поля заголовка. Обращаться к ним можно с помощью универсальных методов `getHeaderFieldKey()` и `getHeaderField()`. Кроме того, для удобства разработчиков предусмотрены перечисленные ниже методы обработки стандартных полей:

   ```
   getContentType
   getContentLength
   getContentEncoding
   getDate
   getExpiration
   getLastModified
   ```

5. Наконец, для доступа к данным указанного ресурса используйте метод `getInputStream()`, который предназначен для получения входного потока. (Это точно такой же входной поток, который возвращается методом `openStream()` класса `URL`.) Существует еще один метод `getContent()`, однако он менее удобен. Для обработки содержимого стандартных типов, например, текста `text/plain` и изображений `image/gif`, потребуется использовать классы из пакета `com.sun`. Кроме того, вы можете зарегистрировать собственные обработчики, но эти способы работы не рассматриваются в данной книге.



Некоторые разработчики, использующие класс `URLConnection`, ошибочно считают, что методы `getInputStream()` и `getOutpuStream()` аналогичны одноименным методам класса `Socket`. Это совсем не так. Класс `URLConnection` способен выполнять много других действий, в частности обрабатывать заголовки запросов и ответов. Поэтому рекомендуется строго придерживаться указанной выше последовательности действий.

Рассмотрим методы класса `URLConnection` более подробно. Существует несколько методов, предназначенных для указания свойств соединения еще до подключения к серверу. Наиболее важными среди них являются методы `setDoInput()` и `setDoOutput()`. По умолчанию соединение получает входной поток для чтения, но не получает выходной поток для записи. Для получения выходного потока (например, для размещения данных на Web-сервере, как показано в следующем разделе) необходимо использовать следующий вызов:

```
connection.setDoOutput(true);
```

Далее можно создать поля заголовка и послать их серверу в составе запроса. Пример заголовка запроса приведен ниже.

```
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=19221888821987
```

Метод `setIfModifiedSince()` предназначен для уведомления соединения о том, что необходимо получить только те данные, которые были изменены после некоторой даты.

Методы `setUseCaches()` и `setAllowUserInteraction()` следует вызывать только в аплетах. Метод `setUseCaches()` указывает браузеру на то, что сначала необходимо проверить кэш. Метод `setAllowUserInteraction()` позволяет аплету использовать показанное на рис. 3.8 диалоговое окно с предложением ввести имя и пароль пользователя для защиты доступа к ресурсам. Однако эти установки не оказывают никакого действия вне аплета.

Наконец, метод `setRequestProperty()` можно использовать для установки пары "имя–значение", имеющей смысл для конкретного протокола. Формат заголовка запроса протокола HTTP описан в документе RFC 2616. Некоторые параметры не очень хорошо документированы, и сведения о них обычно можно получить только у коллег-программистов.

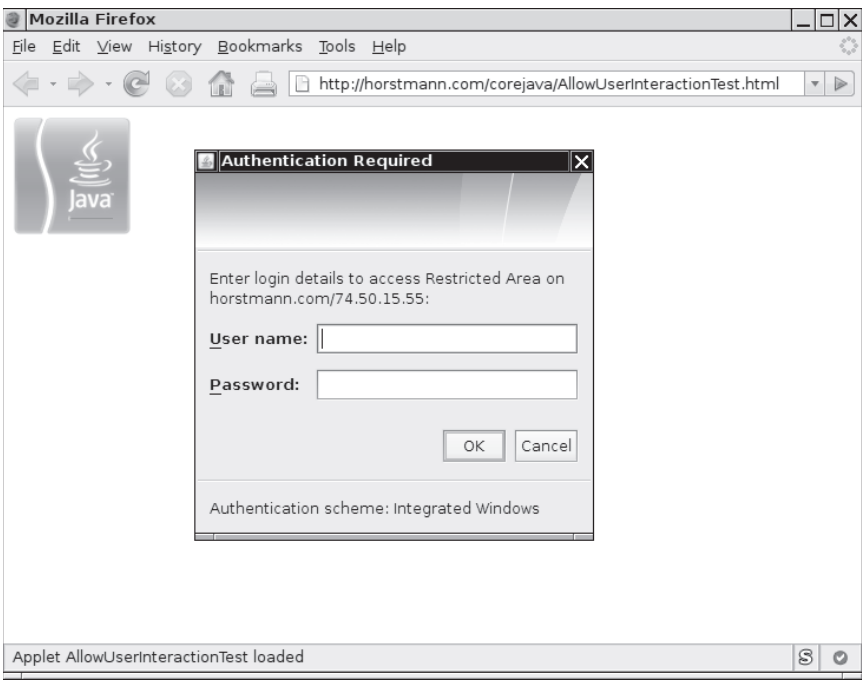


Рис. 3.8. Диалоговое окно для ввода имени и пароля

Например, для доступа к защищенной паролем Web-странице выполните перечисленные ниже действия.

1. Создайте строку, представляющую собой конкатенацию имени пользователя, двоеточия и пароля:

   ```
   String input = username + ":" + password;
   ```

2. Перекодируйте полученную строку согласно алгоритму кодирования base64, который преобразует последовательность байтов в последовательность ASCII-символов:

   ```
   String encoding = base64Encode(input);
   ```

3. Вызовите метод `setRequestProperty()` с именем свойства `"Authorization"` и значением `"Basic"` и `encoding`:

   ```
   connection.setRequestProperty("Authorization", "Basic " + encoding);
   ```



Только что вы познакомились со способом обращения к защищенной паролем Web-странице. Для доступа к защищенному паролем FTP-файлу используется совершенно другой подход. Вам нужно лишь создать URL следующего вида:

```
ftp://username:password@ftp.yourserver.com/pub/file.txt
```

После вызова метода `connect()` можно запросить заголовочную информацию. Рассмотрим сначала способ перечисления всех полей заголовка. Создатели класса посчитали необходимым создать собственный способ перебора полей:

```
String key = connection.getHeaderFieldKey(n);
```

Такой вызов метода позволяет получить *n*-й ключ заголовка, причем нумерация начинается с единицы! Этот вызов возвращает значение `null`, если *n* равно нулю или больше общего количества полей заголовка. Однако для определения числа полей не предусмотрено никакого другого метода. Чтобы перебрать все поля, приходится просто вызывать метод `getHeaderFieldKey()` до тех пор, пока не будет получено значение `null`. Аналогично работает показанный ниже метод получения значения поля:

```
String value = connection.getHeaderField(n);
```

Метод `getHeaderFields()` возвращает объект `Map` с полями заголовка. Более подробно наборы данных типа `Map` рассматриваются в главе 2.

```
Map<String,List<String>> headerFields =
   connection.getHeaderFields();
```

Приведем несколько полей заголовка для типичного HTTP-ответа:

```
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 разработчиков предусмотрено шесть методов получения значений для наиболее распространенных полей заголовка; все они перечислены в табл. 3.1. Методы, которые возвращают значения `long`, начинают отсчет количества секунд с полуночи 1 января 1970 г.

Таблица 3.1. Методы для работы со значениями в полях заголовков запросов

Имя ключа	Метод	Возвращаемый тип
Date	getDate	long
Expires	getExpiration	long
Last-Modified	getLastModified	long
Content-Length	getContentLength	int
Content-Type	getContentType	String
Content-Encoding	getContentEncoding	String

Программа, код которой приведен в листинге 3.7, позволяет экспериментировать с URL-соединениями. Запустив программу, вы можете указать в командной строке URL, имя пользователя и пароль:

```
java URLConnectionTest http://www.yourserver.com user password
```

При успешном выполнении программа отобразит на экране следующие данные:

- все ключи и значения поля заголовка;
- значения, возвращаемые шестью методами, предназначенными для доступа к наиболее распространенным полям, которые перечислены в табл. 3.1;
- первые 10 строк запрашиваемого ресурса.

Это довольно простая программа за исключением той части, в которой выполняются преобразования base64. Вместо использованного в этом примере класса можно применить недокументированный класс `sun.misc.BASE64Encoder`. Достаточно только заменить используемый вызов показанным ниже вызовом другого метода:

```
String encoding =
   new sun.misc.BASE64Encoder().encode(input.getBytes());
```

Однако мы не хотели полагаться на недокументированные классы, поэтому использовали в программе свой класс.



Класс `javax.mail.internet.MimeUtility` в стандартном расширении JavaMail платформы Java также содержит метод кодирования base64. В JDK включен класс `java.util.prefs.Base64`, который не является общедоступным, поэтому вы не можете использовать его в своей программе.

Листинг 3.7. Содержимое файла `URLConnectionTest.java`

```
1.  import java.io.*;
2.  import java.net.*;
3.  import java.util.*;
4.
5.  /**
6.   * Пример соединения с сервером по заданному URL
7.   * и отображения полей заголовка в составе ответа
8.   * с 10 строками запрашиваемых данных.
9.   * Введите в командной строке URL, а также
10.  * (в случае необходимости) имя пользователя и
11.  * пароль (для HTTP-идентификации).
12.  * @version 1.11 2007-06-26
13.  * @author Cay Horstmann
14.  */
```

```
15. public class URLConnectionTest
16. {
17.    public static void main(String[] args)
18.    {
19.       try
20.       {
21.          String urlName;
22.          if (args.length > 0)
23.             urlName = args[0];
24.          else
25.             urlName = "http://java.sun.com";
26.
27.          URL url = new URL(urlName);
28.          URLConnection connection = url.openConnection();
29.
30.          // Установка пользовательского имени и пароля
31.          // (если они указаны в командной строке).
32.
33.          if (args.length > 2)
34.          {
35.             String username = args[1];
36.             String password = args[2];
37.             String input = username + ":" + password;
38.             String encoding = base64Encode(input);
39.             connection.setRequestProperty("Authorization",
40.               "Basic " + encoding);
41.          }
42.          connection.connect();
43.
44.          // Вывод полей заголовка
45.
46.          Map<String, List<String>> headers =
47.             connection.getHeaderFields();
48.          for (Map.Entry<String, List<String>> entry :
49.             headers.entrySet())
50.          {
51.             String key = entry.getKey();
52.             for (String value : entry.getValue())
53.                System.out.println(key + ": " + value);
54.          }
55.
56.          // Вывод значений полей, полученных с помощью
57.          // специальных методов (вместе с именами методов).
58.
59.          System.out.println("----------");
60.          System.out.println("getContentType: " +
61.             connection.getContentType());
62.          System.out.println("getContentLength: " +
63.             connection.getContentLength());
64.          System.out.println("getContentEncoding: " +
65.             connection.getContentEncoding());
66.          System.out.println("getDate: " + connection.getDate());
67.          System.out.println("getExpiration: " +
68.             connection.getExpiration());
69.          System.out.println("getLastModifed: " +
70.             connection.getLastModified());
71.          System.out.println("----------");
```

```
72.         Scanner in = new Scanner(connection.getInputStream());
73.
74.         // Вывод первых десяти строк содержимого.
75.
76.         for (int n = 1; in.hasNextLine() && n <= 10; n++)
77.            System.out.println(in.nextLine());
78.         if (in.hasNextLine()) System.out.println(". . .");
79.       }
80.       catch (IOException e)
81.       {
82.         e.printStackTrace();
83.       }
84.    }
85.
86.    /**
87.     * Кодирование base64 строки.
88.     * @param s Строка
89.     * @return Строка, закодированная base64
90.     */
91.    public static String base64Encode(String s)
92.    {
93.       ByteArrayOutputStream bOut = new ByteArrayOutputStream();
94.       Base64OutputStream out = new Base64OutputStream(bOut);
95.       try
96.       {
97.          out.write(s.getBytes());
98.          out.flush();
99.       }
100.      catch (IOException e)
101.      {
102.      }
103.      return bOut.toString();
104.   }
105. }
106.
107. /**
108.  * Этот фильтр преобразует поток байтов в код,
109.  * полученный на основе алгоритма base64.
110.  * Согласно алгоритму base64 каждые 3 байта преобразуются
111.  * в 4 символа по следующей схеме:
112.  * |11111122|22223333|33444444|
113.  * Каждый набор из 6 битов кодируется в соответствии
114.  * с картой toBase64. Если количество входных данных
115.  * не кратно 3, то последняя группа из 4 символов
116.  * дополняется одним или двумя символами =. Каждая
117.  * выходная строка имеет длину не более 76 символов.
118.  */
119. class Base64OutputStream extends FilterOutputStream
120. {
121.    /**
122.     * Создает фильтр.
123.     * @param out Поток для фильтра
124.     */
125.    public Base64OutputStream(OutputStream out)
126.    {
127.       super(out);
128.    }
```

```
129.   public void write(int c) throws IOException
130.   {
131.      inbuf[i] = c;
132.      i++;
133.      if (i == 3)
134.      {
135.         super.write(toBase64[(inbuf[0] & 0xFC) >> 2]);
136.         super.write(toBase64[((inbuf[0] & 0x03) << 4) |
137.            ((inbuf[1] & 0xF0) >> 4)]);
138.         super.write(toBase64[((inbuf[1] & 0x0F) << 2) |
139.            ((inbuf[2] & 0xC0) >> 6)]);
140.         super.write(toBase64[inbuf[2] & 0x3F]);
141.         col += 4;
142.         i = 0;
143.         if (col >= 76)
144.         {
145.            super.write('\n');
146.            col = 0;
147.         }
148.      }
149.   }
150.
151.   public void flush() throws IOException
152.   {
153.      if (i == 1)
154.      {
155.         super.write(toBase64[(inbuf[0] & 0xFC) >> 2]);
156.         super.write(toBase64[(inbuf[0] & 0x03) << 4]);
157.         super.write('=');
158.         super.write('=');
159.      }
160.      else if (i == 2)
161.      {
162.         super.write(toBase64[(inbuf[0] & 0xFC) >> 2]);
163.         super.write(toBase64[((inbuf[0] & 0x03) << 4) |
164.            ((inbuf[1] & 0xF0) >> 4)]);
165.         super.write(toBase64[(inbuf[1] & 0x0F) << 2]);
166.         super.write('=');
167.            }
168.            if (col > 0)
169.            {
170.                  super.write('\n');
171.                  col = 0;
172.            }
173.      }
174.
175.   private static char[] toBase64 = { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
176.         'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V',
177.         'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
178.         'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x',
179.         'y', 'z', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+', '/' };
180.
181.   private int col = 0;
182.   private int i = 0;
183.   private int[] inbuf = new int[3];
184. }
```



Часто возникает вопрос о том, поддерживается ли на платформе Java доступ к защищенным Web-страницам (URL которых начинается с `https:`). Начиная с Java SE 1.4 средства поддержки SSL (Secure Socket Layer) входят в стандартную библиотеку. Ранее можно было лишь устанавливать SSL-соединения из аплетов, пользуясь средствами SSL, реализованными в браузерах.

API java.net.URL 1.0

- `InputStream openStream()`

  Открывает входной поток для чтения данных ресурса.

- `URLConnection openConnection()`

  Возвращает объект `URLConnection`, который управляет соединением с ресурсом.

API java.net.URLConnection 1.0

- `void setDoInput(boolean doInput)`
- `boolean getDoInput()`

  Если параметр `doInput` имеет значение `true`, пользователь может получать данные посредством текущего объекта `URLConnection`.

- `void setDoOutput(boolean doOutput)`
- `boolean getDoOutput()`

  Если параметр `doOutput` имеет значение `true`, пользователь может передавать данные посредством текущего объекта `URLConnection`.

- `void setIfModifiedSince(long time)`
- `long getIfModifiedSince()`

  Свойство `ifModifiedSince` конфигурирует `URLConnection` для извлечения 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 значение параметра `useCaches` равно `true`, извлечение данных из локального кэша разрешено. Учтите, что объект `URLConnection` не поддерживает кэш. Кэш должен быть предоставлен внешней программой, например, браузером.

- `void setAllowUserInteraction(boolean allowUserInteraction)`
- `boolean getAllowsUserInteraction()`

  Если значение параметра `allowUserInteraction` равно `true`, у пользователя может запрашиваться пароль. Учтите, что объект `URLConnection` не обладает никакими средствами выполнения таких запросов. Запрос пароля должен быть организован во внешней программе, например, в браузере или подключаемом модуле.

- `void setConnectTimeout(int timeout)` 5.0
- `int getConnectTimeout()` 5.0

  Устанавливает или возвращает значение тайм-аута (выраженное в миллисекундах) для соединения. Если время тайм-аута истекло до установления соединения, метод `connect` соответствующего потока ввода генерирует исключение `SocketTimeoutException`.

- `void setReadTimeout(int timeout)` 5.0
- `int getReadTimeout()` 5.0

  Устанавливает или возвращает значение тайм-аута для чтения данных (значение тайм-аута выражается в миллисекундах). Если время тайм-аута истекло до успешного выполнения операции чтения, метод `read` генерирует исключение `SocketTimeoutException`.
- `void setRequestProperty(String key, String value)`

  Устанавливает значение поля заголовка.
- `Map<String,List<String>> getRequestProperties()` 1.4

  Возвращает карту, содержащую свойства запроса. Несколько свойств, соответствующих одному ключу, помещаются в список.
- `void connect()`

  Устанавливает соединение с удаленным ресурсом и получает заголовок ответа.
- `Map<String,List<String>> Map getHeaderFields()` 1.4

  Возвращает карту, содержащую поля заголовка ответа. Все значения, соответствующие одному ключу, организуются в виде карты.
- `String getHeaderFieldKey(int n)`

  Возвращает ключ *n*-го поля заголовка ответа или `null`, если *n* меньше или равно 0 либо превышает количество полей.
- `String getHeaderField(int n)`

  Возвращает значение *n*-го поля заголовка ответа или `null`, если *n* меньше или равно 0 либо 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 тип кодирования содержимого, например, `gzip`. Применяется редко, потому что используемая по умолчанию кодировка не всегда указывается в поле `Content-Encoding`.
- `long getDate()`
- `long getLastModifed()`
- `long getExpiration()`

  Возвращает время создания, последней модификации ресурса либо время, когда срок действия ресурса истекает. Время указывается в секундах, начиная с 1 января 1970 г., по Гринвичу.
- `InputStream openInputStream()`
- `OutputStream openOutputStream()`

  Возвращает поток для чтения ресурса или записи данных.
- `Object getContent()`

  Выбирает соответствующий обработчик содержимого для чтения ресурса. Этот метод вряд ли полезен для чтения данных стандартного типа, например, `text/plain` или `image/gif`, если только разработчику не нужно создать собственный обработчик этих типов.

Отправка данных формы

В предыдущем разделе описывался способ чтения данных с Web-сервера, а здесь рассматривается способ передачи данных из клиентской программы Web-серверу, а также другим программам, которые может вызывать Web-сервер, например, CGI-сценариям и сервлетам.

Для передачи информации от браузера Web-серверу нужно заполнить *форму*, подобную показанной на рис. 3.9.

Когда пользователь щелкает на кнопке Submit (Отправить), данные в полях редактирования, а также информация о состоянии флажков и переключателей передается Web-серверу. Сервер вызывает программу, которая обрабатывает данные, введенные пользователем в форме.

Существует целый ряд технологий, позволяющих Web-серверу вызывать программы. Наиболее часто используются Java-сервлеты, JavaServer Faces, Microsoft ASP (Active Server Pages) и сценарии CGI (Common Gateway Interface). В дальнейшем, говоря о программе, выполняющейся на стороне сервера, мы будем использовать термин *сценарий*, независимо от того, какая технология применяется в конкретном случае.

Сценарий, выполняющийся на стороне сервера, обрабатывает данные, введенные пользователем в форме, и генерирует новую HTML-страницу, которую Web-сервер передает браузеру. Последовательность действий по обработке данных формы условно показана на рис. 3.10. Страница, созданная сервером, может содержать новую информацию (например, результаты поиска) или только подтверждение о получении введенных данных.

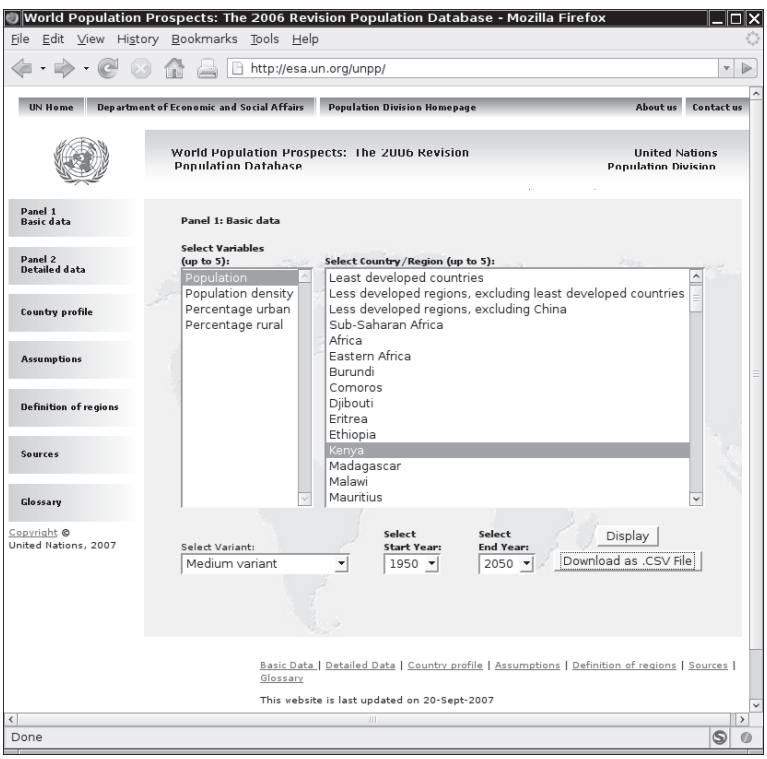


Рис. 3.9. HTML-форма

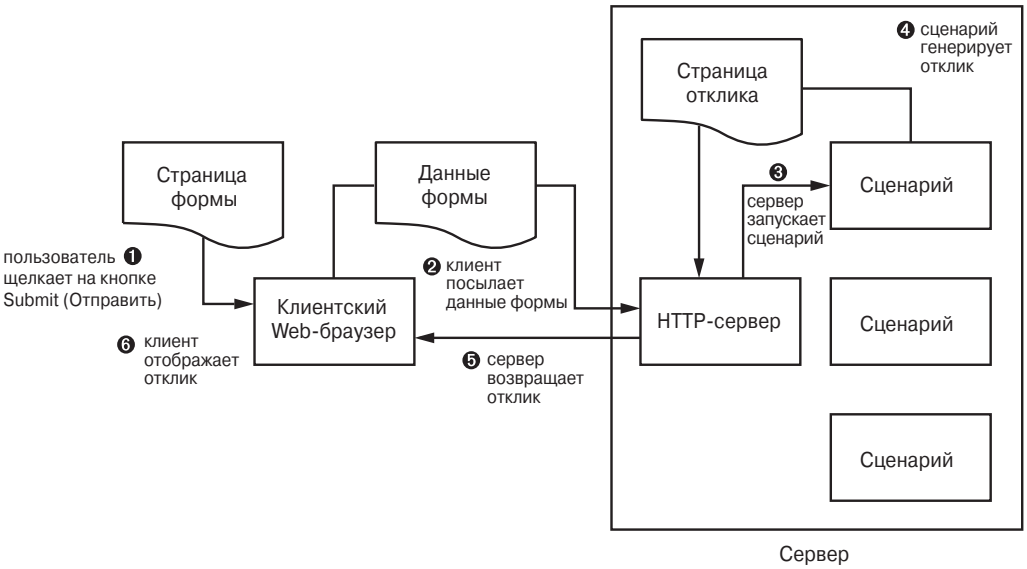


Рис. 3.10. Обработка данных сценарием на стороне сервера

В данной книге мы не будем обсуждать реализацию сценариев, предназначенных для использования на стороне сервера, а ограничимся лишь написанием клиентских программ, предназначенных для взаимодействия с существующими сценариями.

При передаче данных Web-серверу не имеет никакого значения, будет ли использован для их интерпретации CGI-сценарий, сервлет или программа другого типа. Клиент посылает данные Web-серверу в стандартном формате, а Web-сервер сам заботится о том, чтобы найти ту программу, которая сгенерирует ответ.

Передача данных Web-серверу может осуществляться с помощью методов `GET` и `POST`.

При использовании метода `GET` параметры просто указываются в конце URL в показанном ниже формате:

```
http://хост/сценарий?параметры
```

Каждый параметр имеет вид *имя=значение*. Параметры разделяются символами `&`. Значения параметров зашифровываются посредством схемы *шифрования URL*, которая подчинена следующим правилам:

- символы от `A` до `Z`, от `a` до `z`, от `0` до `9`, а также символы `. - * _` остаются неизмененными;
- все пробелы заменяются символами `+`;
- все остальные символы зашифровываются в UTF-8, а каждый байт преобразуется в форму `%` , за которым следует двузначное шестнадцатеричное число.

Например, для передачи названия улицы *S. Main* нужно ввести символы `S%2e+Main`. Здесь шестнадцатеричное число `2e` (или десятичное `46`) представляет собой ASCII-код точки.

Благодаря такому шифрованию, промежуточные программы не будут путаться в пробелах и смогут правильно интерпретировать другие символы.

Например, во время написания этой книги Web-сервер Yahoo! содержал сценарий `py.maps.py` на хосте `maps.yahoo.com`. Сценарию необходимы два параметра: `addr` и `csz`.

Чтобы получить карту с адресом *1 Infinite Loop, Cupertino, CA*, нужно сформировать следующий URL:

```
http://maps.yahoo.com/py/maps.py?addr=1+Infinite+Loop&csz=Cupertino+CA
```

Метод GET очень прост в применении, но имеет существенный недостаток: в большинстве браузеров количество символов, которые можно включить в GET-запрос, ограничено.

В методе POST не нужно включать параметры в URL. Вместо этого необходимо получить выходной поток из объекта URLConnection и записать в него пары "имя–значение". При этом необходимо также выполнить URL-кодирование значений, разделив их символом &.

Рассмотрим этот процесс более подробно. Для передачи данных сценарию нужно сначала создать объект URLConnection:

```
URL url = new URL("http://компьютер/сценарий");
URLConnection connection = url.openConnection();
```

Затем необходимо вызвать метод setDoOutput() для того, чтобы разрешить передачу данных через соединение:

```
connection.setDoOutput(true);
```

После этого с помощью метода getOutputStream() потребуется получить выходной поток. Для передачи текстовых данных удобно инкапсулировать поток в объекте PrintWriter.:

```
PrintWriter out = new PrintWriter(connection.getOutputStream());
```

Теперь можно передавать данные серверу:

```
out.print(name1 + "=" + URLEncoder.encode(value1, "UTF-8") + "&");
out.print(name2 + "=" + URLEncoder.encode(value2, "UTF-8"));
```

После передачи данных нужно закрыть выходной поток:

```
out.close();
```

И, наконец, необходимо вызывать метод getInputStream() и прочитать ответ сервера.

Рассмотрим следующий пример. Web-сайт http://esa.un.org/unpp содержит страницу с формой для ввода запроса на получение статистических сведений о численности и структуре населения (см. рис. 3.9). В HTML-коде формы содержится следующий дескриптор:

```
<form action="p2k0data.asp" method="post">
```

В этом дескрипторе указан метод отправки данных (POST) и сценарий p2k0data.asp, который будет выполнен после того, как пользователь щелкнет на кнопке Submit, предназначенной для отправки введенных данных.

Далее нужно найти имена полей, которые потребуются указанному сценарию. Рассмотрим внимательно элементы формы. Каждый из них имеет атрибут name:

```
<select name="Variable"
<option value="12;">Population</option>
...другие параметры...
</select>
```

В этом примере поле имеет имя Variable. Оно содержит названия таблиц с разными статистическими данными. Например, таблица с параметром "12; " содержит ежегодные данные по общей численности населения. Далее в HTML-коде этой формы

ввода можно найти поле Location с параметрами, которые обозначают сокращенные названия стран, например 900 (весь мир) и 404 (Кения).

Нам нужно задать еще некоторые поля. Чтобы получить оценку численности населения в Кении в период с 1950 по 2050 гг., понадобится следующая строка:

```
Panel=1&Variable=12%3b&Location=404&Varient=2&StartYear=1950&EndYear=2050&
  DoWhat=Download+as+%2eCSV+File
```

Затем ее необходимо послать Web-серверу по адресу:

```
http://esa.un.org/unpp/p2k0data.asp
```

После получения и обработки этих параметров сценарий пошлет 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. Именно поэтому он был выбран для данного примера, чтобы ознакомиться с основными принципами работы сценариев, не отвлекаясь на изучение HTML-дескрипторов.

В листинге 3.8 приведен код программы, которая с помощью метода POST посылает данные серверу. Для выбора страны и просмотра результатов используется простой интерфейс (рис. 3.11).

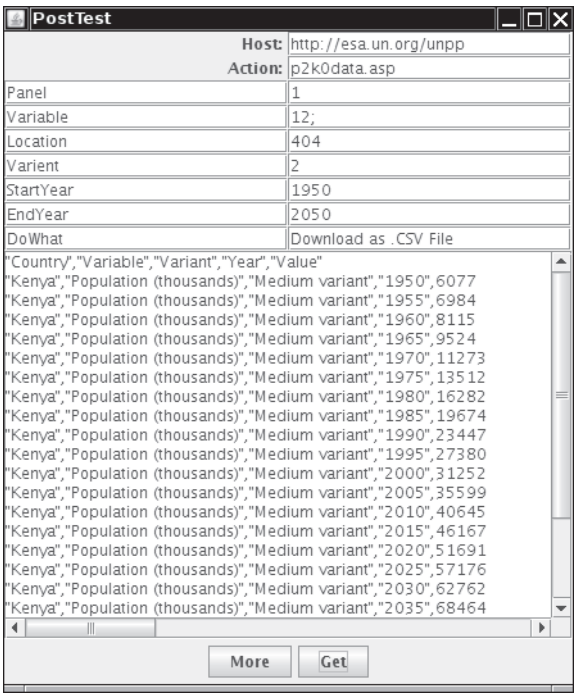


Рис. 3.11. Получение информации с сервера

В методе `doPost()` мы сначала создаем соединение, затем вызываем метод `setDoOutput(true)`, после чего открываем выходной поток. Затем мы перебираем имена и значения объекта `Map` и для каждой пары "имя–значение" передаем *имя* `=` *значение*, отделяя одну пару от другой символом `&`.

```
out.print(name);
out.print('=');
out.print(URLEncoder.encode(value, "UTF-8"));
if (если есть другие пары "поле-значение") out.print('&');
```

И, наконец, мы считываем ответ сервера. При этом следует учитывать одну особенность. Если при работе сценария возникнет ошибка, то в таком случае вызов метода `connection.getInputStream()` приведет к возникновению исключения `FileNotFoundException`. Однако сервер продолжит передачу данных, а именно — пошлет HTML-страницу с сообщением об ошибке. Обычно это сообщение `"Error 404-page not found"` о том, что данная страница не найдена. Для перехвата этой страницы нужно привести объект `URLConnection` к типу `HttpURLConnection` и вызвать метод `getErrorStream()`:

```
InputStream err = (HttpURLConnection) connection).getErrorStream();
```

Больше с любознательной, нежели с практической целью, вам может понадобиться знать, какую именно информацию `URLConnection` отправляет на сервер в дополнение к введенным вами данным.

Объект `URLConnection` сначала отправляет на сервер заголовок запроса. Во время отправки данных формы заголовок содержит следующую информацию:

```
Content-Type: application/x-www-form-urlencoded
```

Заголовок для `POST` должен включать также сведения о длине содержимого; например:

```
Content-Length: 124
```

Конец заголовка отмечается пустой строкой. Затем следует порция данных. Web-сервер отделяет заголовок и направляет порцию данных сценарию на стороне сервера.

Обратите внимание, что объект `URLConnection` буферизирует все данные, которые вы посылаете в выходной поток, поскольку он должен сначала определить общую длину содержимого.

Используемый в этой программе подход очень полезен для создания запросов на получение информации с Web-сервера. Для этого нужно, просмотрев HTML-код Web-страницы, генерирующей аналогичный запрос, узнать необходимые параметры и удалить из ответа все лишние HTML-дескрипторы и другую ненужную информацию.

Листинг 3.8. Содержимое файла `PostTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.io.*;
4.  import java.net.*;
5.  import java.util.*;
6.  import javax.swing.*;
7.
8.  /**
9.   * Данная программа демонстрирует использование
10.  * класса URLConnection для передачи запроса POST.
11.  *version 1.20 2007-06-25
12.  *@author
13.  */
```

```
14. public class PostTest
15. {
16.    public static void main(String[] args)
17.    {
18.       EventQueue.invokeLater(new Runnable()
19.          {
20.             public void run()
21.             {
22.                JFrame frame = new PostTestFrame();
23.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24.                frame.setVisible(true);
25.             }
26.          });
27.    }
28. }
29. class PostTestFrame extends JFrame
30. {
31.    /**
32.     * Создает запрос POST и возвращает ответ сервера.
33.     * @param urlString URL, по которому происходит передача
34.     * @param nameValuePairs Карта с данными для запроса
35.     * @return Ответ сервера (получен либо из входного потока,
36.     * либо из потока ошибки)
37.     */
38.    public static String doPost(String urlString,
39.       Map<String, String> nameValuePairs) throws IOException
40.    {
41.       URL url = new URL(urlString);
42.       URLConnection connection = url.openConnection();
43.       connection.setDoOutput(true);
44.       PrintWriter out = new PrintWriter(connection.getOutputStream());
45.       boolean first = true;
46.       for (Map.Entry<String, String> pair :
47.          nameValuePairs.entrySet())
48.       {
49.          if (first) first = false;
50.          else out.print('&');
51.          String name = pair.getKey();
52.          String value = pair.getValue();
53.          out.print(name);
54.          out.print('=');
55.          out.print(URLEncoder.encode(value, "UTF-8"));
56.       }
57.       out.close();
58.       Scanner in;
59.       StringBuilder response = new StringBuilder();
60.       try
61.       {
62.          in = new Scanner(connection.getInputStream());
63.       }
64.       catch (IOException e)
65.       {
66.          if (!(connection instanceof HttpURLConnection)) throw e;
67.          InputStream err = ((HttpURLConnection)connection).getErrorStream();
68.          if (err == null) throw e;
69.          in = new Scanner(err);
70.       }
```

```
71.       while (in.hasNextLine())
72.       {
73.          response.append(in.nextLine());
74.          response.append("\n");
75.       }
76.
77.       in.close();
78.       return response.toString();
79.    }
80.
81.    public PostTestFrame()
82.    {
83.       setTitle("PostTest");
84.
85.       northPanel = new JPanel();
86.       add(northPanel, BorderLayout.NORTH);
87.       northPanel.setLayout(new GridLayout(0, 2));
88.       northPanel.add(new JLabel("Host: ", SwingConstants.TRAILING));
89.       final JTextField hostField = new JTextField();
90.       northPanel.add(hostField);
91.       northPanel.add(new JLabel("Action: ", SwingConstants.TRAILING));
92.       final JTextField actionField = new JTextField();
93.       northPanel.add(actionField);
94.       for (int i = 1; i <= 8; i++)
95.          northPanel.add(new JTextField());
96.
97.       final JTextArea result = new JTextArea(20, 40);
98.       add(new JScrollPane(result));
99.
100.      JPanel southPanel = new JPanel();
101.      add(southPanel, BorderLayout.SOUTH);
102.      JButton addButton = new JButton("More");
103.      southPanel.add(addButton);
104.      addButton.addActionListener(new ActionListener()
105.         {
106.            public void actionPerformed(ActionEvent event)
107.            {
108.               northPanel.add(new JTextField());
109.               northPanel.add(new JTextField());
110.               pack();
111.            }
112.         });
113.
114.         JButton getButton = new JButton("Get");
115.         southPanel.add(getButton);
116.         getButton.addActionListener(new ActionListener()
117.            {
118.               public void actionPerformed(ActionEvent event)
119.               {
120.                  result.setText("");
121.                  final Map<String, String> post = new HashMap<String, String>();
122.                  for (int i = 4; i < northPanel.getComponentCount(); i += 2)
123.                  {
124.                     String name = ((JTextField) northPanel.getComponent(i)).getText();
125.                     if (name.length() > 0)
126.                  {
127.                     String value =
128.                           ((JTextField) northPanel.getComponent(i + 1)).getText();
```

```
129.                  post.put(name, value);
130.               }
131.            }
132.            new SwingWorker<Void, Void>()
133.               {
134.                  protected Void doInBackground() throws Exception
135.                  {
136.                     try
137.                     {
138.                        String urlString =
139.                           hostField.getText() + "/" + actionField.getText();
140.                        result.setText(doPost(urlString, post));
141.                     }
142.                     catch (IOException e)
143.                     {
144.                        result.setText("" + e);
145.                     }
146.                     return null;
147.                  }
148.               }.execute();
149.         }
150.       });
151.     pack();
152.   }
153.   private JPanel northPanel;
154. }
```

API java.net.HttpURLConnection 1.0

- `InputStream getErrorStream()`

  Возвращает поток, из которого считываются сообщения Web-сервера об ошибках.

API java.net.URLEncoder 1.0

- `static String encode(String s, String encoding)` 1.4

  Возвращает строку `s` в URL-кодированном виде. Кодирование проводится согласно указанной схеме. В URL-кодировке символы `A–Z`, `a–z`, `0–9`, `-`, `_`, `.` и `*` остаются неизменными. Пробелы передаются в виде символов `+`, а все остальные символы — в виде `%UV`, где `UV` — шестнадцатеричное представление байта.

API java.net.URLDecoder 1.2

- `static string decode(String s, String encoding)` 1.4

  Возвращает URL-кодированную строку `s` в декодированном виде. Декодирование проводится согласно указанной схеме.

В настоящей главе было показано, как пишется код для сетевых клиентов и серверов на языке Java, и как происходит сбор информации с Web-серверов. В следующей главе мы поговорим о связности (возможности соединения) баз данных. Вы узнаете, как работать с реляционными базами данных на Java с использованием JDBC API. Следующая глава содержит также краткое введение в иерархические базы данных (такие как каталоги LDAP) и JNDI API.

Глава 4

РАБОТА С БАЗАМИ ДАННЫХ

В ЭТОЙ ГЛАВЕ...

- Структура JDBC
- Язык SQL
- Конфигурирование JDBC
- Выполнение операторов SQL
- Выполнение запросов
- Прокручиваемые и обновляемые результирующие наборы
- Наборы строк
- Метаданные
- Транзакции
- Управление соединениями в Web- и производственных приложениях
- Введение в LDAP

Летом 1996 г. компания Sun выпустила первую версию интерфейса для организации доступа Java-приложений к базам данных JDBC (Java DataBase Connectivity). Этот интерфейс позволяет программистам 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 превратился в один из наиболее распространенных интерфейсов в библиотеке Java.

Средства JDBC неоднократно обновлялись. В состав JDK 1.2, выпущенного в 1998 г., была включена версия JDBC 2. JDBC 3 появился в версиях Java SE 1.4 и 5.0. Во время публикации данной книги последней версией была JDBC 4, входящая в состав Java SE 6.

В этой главе мы представим идеи, лежащие в основе JDBC. Мы познакомим вас (а для некоторых это будет просто напоминанием) с языком SQL, который является стандартным средством доступа к реляционным базам данных. Затем будут приведены примеры использования интерфейса JDBC, демонстрирующие наиболее распространенные способы работы с базами данных. Завершит эту главу краткое введение в иерархические базы данных, протокол LDAP и средства JNDI (Java Naming and Directory Interface — интерфейс именования и каталогов Java).



Как заявляют в Sun, JDBC — это термин для представления торговой марки, а не акроним Java Database Connectivity. Он был назван на манер устаревшего термина ODBC — стандартного интерфейса для работы с базами данных, который был предложен компанией Microsoft, и был переведен в стандарт SQL.

Структура JDBC

С самого начала разработчики Java в компании Sun понимали потенциальные преимущества данного языка при работе с базами данных. С 1995 г. они начали работу по расширению стандартной библиотеки Java для организации взаимодействия с языком SQL и доступа к базам данных. Сначала они попробовали создать такие расширения Java, которые позволили бы осуществлять доступ к произвольной базе данных только средствами Java. Очень скоро разработчики убедились в бесперспективности такой постановки задачи. Кроме того, поставщики программного обеспечения для баз данных всеми силами стремились помочь сотрудникам в деле создания стандартного сетевого протокола доступа к базам данных, но при том условии, если за основу будет принят *их собственный* протокол.

В конечном итоге поставщики СУБД и инструментов доступа к базам все-таки согласились с тем, чтобы сотрудники Sun создали Java API для SQL-доступа к данным и диспетчер драйверов, который позволил бы подключать к базам драйверы независимых производителей. Такой подход позволял производителям СУБД создавать собственные драйверы, которые подключались бы с помощью диспетчера. Предполагалось, что это будет простой механизм регистрации драйверов. В результате было создано два интерфейса. Разработчики приложений используют JDBC API, а поставщики баз данных и инструментальных средств — JDBC Driver API.

Подход, применяемый при создании JDBC, основан на очень успешной модели ODBC-интерфейса компании Microsoft. В основе JDBC и ODBC лежит общий механизм: программы, соответствующие API, могут взаимодействовать с диспетчером драйверов JDBC, который, в свою очередь, использует подсоединенные драйверы для организации взаимодействия с базой данных.

Сказанное означает, что для работы с базами данных достаточно использовать JDBC API (рис. 4.1).

Список имеющихся JDBC-драйверов можно найти по адресу `http://developers.sun.com/products/jdbc/drivers`.

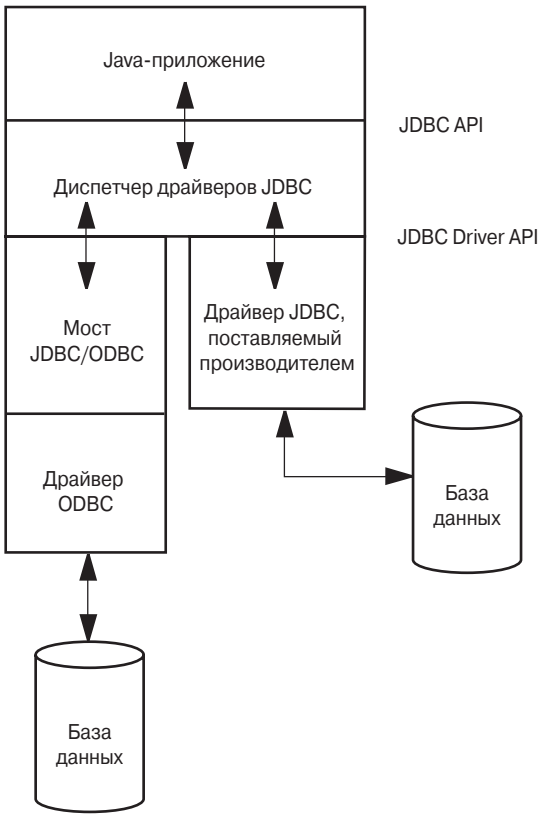


Рис. 4.1. Взаимодействие JDBC и базы данных

Типы JDBC-драйверов

Каждый JDBC-драйвер принадлежит одному из перечисленных ниже *типов*.

- *Драйвер типа 1.* Транслирует JDBC в ODBC и для взаимодействия с базой данных использует драйвер ODBC. Компания Sun включила в состав JDK один такой драйвер — *мост JDBC/ODBC*. Однако для его использования требуется соответствующим образом установить и конфигурировать ODBC-драйвер. В первом выпуске JDBC этот мост предполагалось использовать только для тестирования, а не для рабочего применения. В настоящее время уже имеется большое количество более удачных драйверов.
- *Драйвер типа 2.* Создается преимущественно на языке Java и частично на собственном языке программирования, который используется для взаимодействия с клиентским API базы данных. Для использования такого драйвера нужно помимо библиотеки Java установить специфический для данной платформы код.

- *Драйвер типа 3*. Создается только на основе библиотеки Java, в которой используется независимый от базы данных протокол взаимодействия сервера и базы. Этот протокол позволяет транслировать запросы в соответствии со спецификой конкретной базы. Если код, зависящий от базы данных, находится только на сервере, развертывание существенно упрощается.
- *Драйвер типа 4*. Представляет собой библиотеку Java, которая транслирует JDBC-запросы непосредственно в протокол конкретной базы данных.

Большинство поставщиков баз данных применяют драйверы типа 3 или 4. Кроме того, некоторые компании специализируются на создании драйверов, которые позволяют добиться более полного соответствия принятым стандартам, поддерживают бо́льшее количество платформ, обладают более высокой производительностью или надежностью, чем драйверы, предлагаемые производителями СУБД.

Основные цели интерфейса JDBC можно сформулировать следующим образом.

- Разработчики создают программы на языке Java, пользуясь для доступа к базам данных стандартными средствами языка SQL (или его специализированными расширениями); при этом они следуют только соглашениям языка Java.
- Производители СУБД и инструментальных средств поставляют драйверы только низкого уровня.



На конференции JavaOne в мае 1996 г. представители Sun указали ряд причин отказа от модели ODBC.

- Модель ODBC трудна в изучении.
- Модель ODBC имеет всего несколько команд с большим количеством параметров. Однако стиль программирования на языке Java основан на использовании большого количества простых и интуитивно понятных методов.
- Модель ODBC основана на использовании указателей типа `void*` и других компонентов языка C, которые отсутствуют в языке Java.
- Модель ODBC менее защищена и труднее для распространения, чем решения, основанные только на Java.

Типичные примеры использования JDBC

Согласно традиционной модели клиент/сервер, графический пользовательский интерфейс реализуется на стороне клиента, а база данных располагается на стороне сервера (рис. 4.2). В этом случае драйвер JDBC находится на стороне клиента.

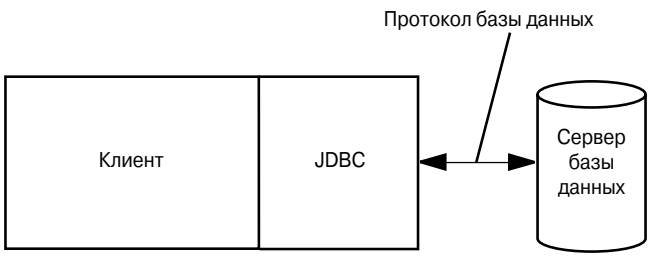


Рис. 4.2. Традиционная структура приложения клиент/сервер

Однако современная тенденция развития программного обеспечения заключается в переходе от архитектуры клиент/сервер к "трехуровневой модели" или даже более совершенной "*n*-уровневой модели". В трехуровневой модели клиент не формирует обращений к базе данных. Вместо этого он обращается к средствам промежуточного уровня на сервере, который, в свою очередь, выполняет запросы к базе данных. Трехуровневая модель имеет два преимущества. Она отделяет *визуальное представление* (на компьютере-клиенте) от *бизнес-логики* (промежуточный уровень) и *данных* (в базе данных). Таким образом, становится возможным доступ к тем же данным и тем же бизнес-правилам посредством клиентов различных типов, например, Java-приложений, аплетов или Web-форм.

Взаимодействие между клиентом и промежуточным уровнем может быть реализовано на основе протокола HTTP (при использовании Web-браузера в качестве клиента) или с помощью какого-то другого механизма, например, посредством RMI (Remote Method Invocation — удаленный вызов методов) — об этом будет рассказано в главе 10. JDBC используется для управления взаимодействием между промежуточным уровнем и базой данных. На рис. 4.3 схематически показана базовая архитектура трехуровневой модели. Следует иметь в виду, что у этой модели существует множество вариантов. В частности, в Java Enterprise Edition определена структура *серверов приложений*, которые управляют компонентами *Enterprise JavaBeans* и предоставляют такие важные услуги, как балансировка нагрузки, кэширование запросов, обеспечение безопасности и отображение на основе объектов. В этой архитектуре JDBC играет важную роль передачи сложных запросов к базе данных. (Более подробную информацию по этой теме можно найти по адресу `http://java.sun.com/javaee`.)

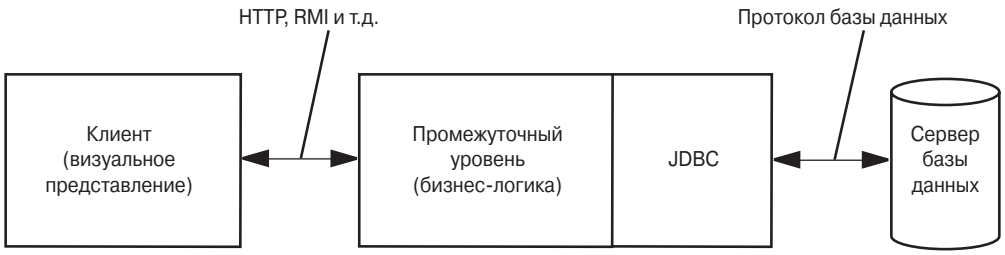


Рис. 4.3. Приложение, созданное в соответствии с трехуровневой моделью



Средства JDBC можно использовать 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, с которого аплет был скопирован. Это означает, что база данных должна быть расположена на том же компьютере, что и Web-сервер (или компонент ретрансляции драйвера типа 3), но системные администраторы редко поступают так. Для того чтобы решить эту проблему, необходимо использовать цифровую подпись.

Язык SQL

Интерфейс JDBC позволяет взаимодействовать с базами данных с помощью языка SQL, который, в свою очередь, образует интерфейс для большинства современных реляционных баз данных. Настольные СУБД предоставляют графический интерфейс, который позволяет пользователям непосредственно манипулировать данными, но доступ к серверным базам данных возможен только с помощью SQL.

Пакет JDBC можно рассматривать лишь как API для взаимодействия с SQL-командами доступа к базам данных. В этом разделе приводится краткое описание SQL. Если вы ранее не имели дела с SQL, этой информации может быть недостаточно. Для более тщательного изучения основ SQL можно порекомендовать книгу Алана Билю (Alan Beaulieu) *Learning SQL* (O'Reilly, 2005 г.) или классическое издание К. Дж. Дейта (C. J. Date) и Хью Дарвена (Hugh Darwen) *A Guide to the SQL Standard* (Addison-Wesley, 1997 г.).

База данных представляет собой набор именованных таблиц со строками и столбцами. Каждый столбец (или поле) имеет *имя*, а данные хранятся в строках.

В качестве примера базы данных рассмотрим набор таблиц (табл. 4.1–4.4), описывающих классические книги по компьютерной тематике.

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. 4.4 представлен внешний вид таблицы `Books`, а на рис. 4.5 — результат *соединения* таблиц `Books` и `Publishers`. Обе таблицы содержат идентификатор издателя. При соединении таблиц по этому коду *в результате запроса* получится таблица, которая содержит данные из обеих таблиц. В каждой строке содержится информация о книге, название и адрес сервера издательства. Обратите внимание на то, что информация с названием и адресом сервера несколько раз дублируется, так как у нас появилось несколько строк, которые относятся к одному издательству.

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 URL-адресом сайта издательства. Но в таком случае данные будут дублироваться уже не на уровне результатов запроса, а в самой базе данных.

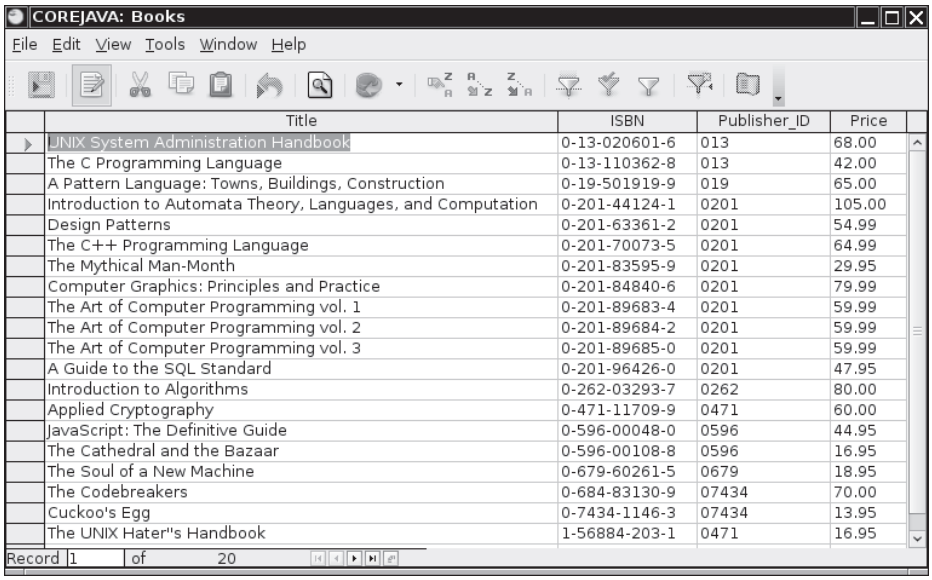


Рис. 4.4. Таблица с данными о книгах

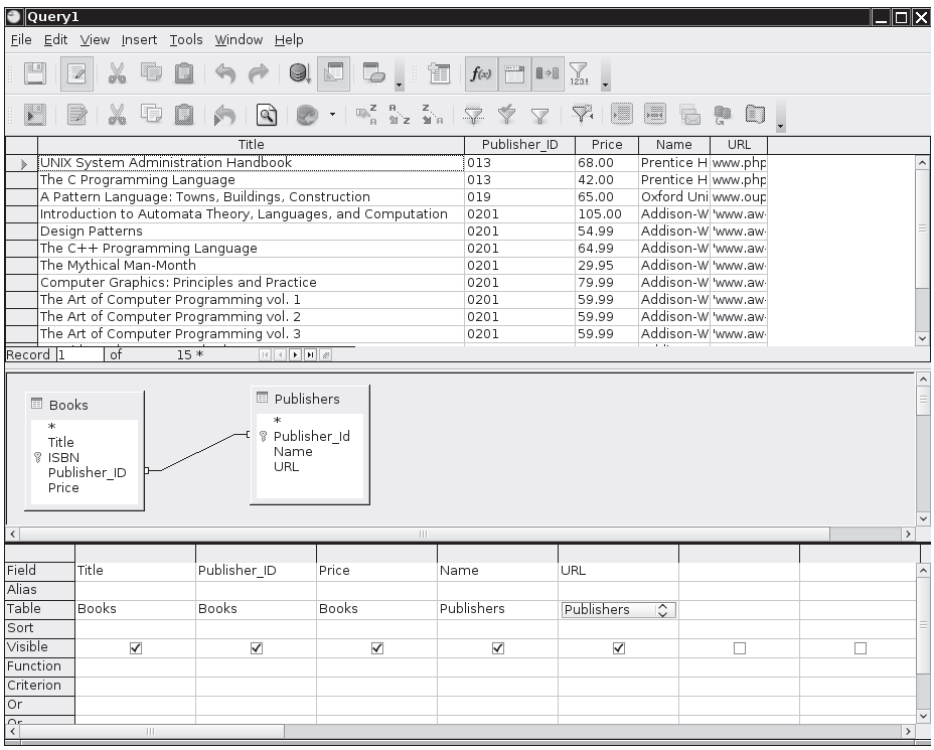


Рис. 4.5. Результат соединения двух таблиц

При изменении URL-адреса сайта придется изменить эти данные во *всех* записях. Ясно, что при выполнении такой трудоемкой задачи могут легко возникнуть ошибки. В реляционной модели данные распределяются среди нескольких таблиц таким образом, чтобы никакая информация не дублировалась без необходимости. Например, URL-адрес сайта каждого издательства хранится в единственном экземпляре в таблице с данными об издательствах. В случае надобности данные из разных таблиц можно легко соединить.

На рис. 4.4 и 4.5 показан инструмент с графическим интерфейсом, предназначенный для просмотра и связывания таблиц. Многие поставщики программного обеспечения предлагают разнообразные интерактивные инструменты создания запросов путем манипуляций со столбцами и ввода данных в готовые формы. Такие средства называются инструментами *создания запросов по образцу* (query by example – QBE). При использовании SQL запрос создается в текстовом виде в строгом соответствии с синтаксисом языка:

```
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 данного раздела описываются основные способы создания таких текстовых запросов. Читатели, знакомые с SQL, могут пропустить этот материал.

По соглашению ключевые слова SQL записываются в верхнем регистре, хотя это правило не является обязательным.

Команда `SELECT` сможет применяться в разных целях, например для выбора всех элементов таблицы `Books`, как показано ниже:

```
SELECT * FROM Books
```

Конструкция `FROM` является обязательной в каждой команде `SELECT`. Она сообщает базе данных о тех таблицах, в которых нужно выполнить поиск данных.

В команде `SELECT` можно указать любые нужные столбцы:

```
SELECT ISBN, Price, Title
FROM Books
```

Выбор строк можно ограничить с помощью условия, указанного посредством конструкции `WHERE`:

```
SELECT ISBN, Price, Title
FROM Books
WHERE Price <= 29.95
```

Следует иметь в виду, что для сравнения в SQL используются операции = и <>, а не == или != (как при программировании на языке Java).



Некоторые поставщики баз данных используют операцию != для указания сравнения, но учтите, что это не соответствует стандарту SQL, а потому полагаться на данную возможность не рекомендуется.

В конструкции `WHERE` может присутствовать операция `LIKE`, посредством которой задаются символы подстановки. В отличие от привычных * и ?, знак % обозначает любое количество символов, а знак _ – один символ. Ниже приведен пример использования операции `LIKE`:

```
SELECT ISBN, Price, Title
FROM Books
WHERE Title NOT LIKE '%n_x%'
```

Здесь выбираются книги, названия которых не содержат таких слов, как UNIX или Linux.

Обратите внимание, что строки заключены в одинарные, а не в двойные кавычки. Одинарная кавычка внутри строки обозначается парой одинарных кавычек, например, так, как показано ниже:

```
SELECT Title
FROM Books
WHERE Books.Title LIKE '%''%'
```

В этом примере требуется выполнить поиск всех названий книг, которые содержат одинарную кавычку.

Чтобы выбрать данные из нескольких таблиц, нужно перечислить их так, как показано ниже:

```
SELECT * FROM Books, Publishers
```

Без конструкции WHERE этот запрос не представляет большого интереса, поскольку позволяет получить *все сочетания* строк из обеих таблиц. В нашем случае таблица Books содержит 20 строк, а таблица Publishers — 8 строк. Поэтому результат выполнения такого запроса будет содержать 20×8 строк с бо́льшим количеством дублирующихся данных. Предположим, что нужно найти только те книги, которые выпущены издательствами, перечисленными в таблице Publishers. Для поиска такого *соответствия* между книгами и издательствами можно использовать показанный ниже запрос:

```
SELECT * FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

Результат выполнения этого запроса содержит 20 строк, т.е. по одной строке для каждой книги, так как издательство, соответствующее каждой книге, упомянуто в таблице Publishers.

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 столбец Publisher_Id из таблицы Publishers). Если такое упоминание столбца приводит к неоднозначному толкованию, для него следует использовать префикс в виде имени таблицы (например, Books.Publisher_Id).

SQL можно применять и для изменения данных в базе данных. Например, предположим, что вы хотели бы сократить на 5 долларов текущую цену для всех книг, в названиях которых содержатся символы C++:

```
UPDATE Books
SET Price = Price - 5.00
WHERE Title LIKE '%C++%'
```

Точно так же, чтобы удалить все книги по C++, понадобится запрос DELETE:

```
DELETE FROM Books
WHERE Title LIKE '%C++%'
```

Наконец, в SQL предусмотрены встроенные функции для вычисления средних значений, поиска максимальных и минимальных значений в столбце и выполнения многих других действий. Более подробная информация доступна по адресу http://sqlzoo.net. (На этом сайте также есть отличные интерактивные обучающие материалы по языку SQL.)

Для вставки новых данных в таблицу используется команда INSERT:

```
INSERT INTO Books
VALUES ('A Guide to the SQL Standard', '0-201-96426-0', '0201', 47.95)
```

Для вставки каждой строки приходится применять отдельную команду INSERT.

Прежде чем создавать запросы, изменять и вставлять данные, необходимо предоставить место для их хранения, т.е. создать таблицу. Для создания новой таблицы используется команда CREATE TABLE, в которой указываются имена и типы данных каждого столбца:

```
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 языка SQL.

Таблица 4.5. Типы данных SQL

Тип данных	Описание
INTEGER или INT	Обычно 32-битное целое число
SMALLINT	Обычно 16-битное целое число
NUMERIC(m,n), DECIMAL(m,n) или DEC(m,n)	Десятичное число с фиксированной точкой, содержащее *m* цифр, в том числе *n* знаков после точки
FLOAT(n)	Число с плавающей точкой с *n* знаками точности
REAL	Обычно 32-битное число с плавающей точкой
DOUBLE	Обычно 64-битное число с плавающей точкой
CHARACTER(n) или CHAR(n)	Строка с фиксированной длиной *n* символов
VARCHAR(n)	Строка переменной длины; максимальная длина *n* символов
BOOLEAN	Логическое значение
DATE	Календарная дата (зависит от реализации)
TIME	Время, зависит от реализации
TIMESTAMP	Дата и время (зависит от реализации)
BLOB	Большой двоичный объект
CLOB	Большой символьный объект

Дополнительные конструкции, задающие ключи и ограничения, которые используются вместе с командой CREATE TABLE, в этой книге не рассматриваются.

Конфигурирование JDBC

Разумеется, вам потребуется СУБД, совместимая с JDBC. Вы можете, например, выбрать IBM DB2, Microsoft SQL Server, MySQL, Oracle или PostgreSQL.

Далее необходимо создать экспериментальную базу данных, например, под названием COREJAVA. Сформируйте новую базу или попросите администратора создать ее, а также наделить вас правами для создания, обновления и удаления таблиц.

Если ранее вам никогда не приходилось инсталлировать базу данных с архитектурой клиент/сервер, то процесс ее установки, конечно, покажется очень сложным, а обнаружить причину возможной неудачи будет довольно трудно. Поэтому в таких случаях рекомендуется прибегнуть к услугам опытных специалистов.

Если вы не имеете никакого опыта работы с базами данных, рекомендуем установить базу Apache Derby, которая является частью некоторых версий JDK 6. (Если ваш JDK не включает ее, загрузить Apache Derby можно по адресу `http://db.apache.org.derby`.)



Компания Sun ссылается на версию Apache Derby, включенную в JDK, как на JavaDB. Во избежание недоразумений, в этой главе мы будем называть ее Derby.

Прежде чем вы сможете написать свою первую СУБД, придется изучить многие вопросы, рассмотрению которых будут посвящены следующие разделы.

URL базы данных

Для установления соединения с базой данных необходимо указать ряд параметров, имеющих отношение к базам данных. Например, это могут быть имена хостов, номера портов и имена баз данных.

В JDBC используется синтаксис описания источника данных, подобный обычным URL:

```
jdbc:derby://localhost:1527/COREJAVA;create=true
jdbc:postgresql:COREJAVA
```

Эти URL определяют базы данных Derby и PostgreSQL по имени `COREJAVA`. Общие правила формирования URL выглядят так:

```
jdbc:название_подпротокола:другие_сведения
```

Здесь *`название_подпротокола`* необходимо для выбора специального драйвера для соединения с базой данных.

Формат представления параметров в части *`другие_сведения`* зависит от используемого подпротокола. Более подробную информацию об этом можно получить в документации, поставляемой производителем СУБД.

JAR-файлы драйверов

Вам необходимо получить JAR-файл, в котором находится драйвер для вашей базы данных. Если вы пользуетесь Derby, понадобится файл `derbyclient.jar`. Если это другая база данных, нужно будет найти подходящий драйвер. К примеру, драйверы PostgreSQL доступны на сайте `http://jdbc.postgresql.org`.

При запуске программы, обращающейся к базе данных, нужно добавить JAR-файл драйвера в параметр `classpath`. (Для компиляции JAR-файл не нужен.)

Для запуска программ из командной строки воспользуйтесь следующей простой командой:

```
Java -classpath .:Jar-файл_драйвера ИмяПрограммы
```

В ОС Windows для отделения текущего каталога (который обозначается символом `.`) от местонахождения JAR-файла драйвера используется точка с запятой.

Запуск базы данных

Прежде чем вы сможете подключиться к серверу базы данных, его нужно запустить. Детали этого процесса зависят от разновидности вашей базы данных.

Для базы данных Derby необходимо выполнить перечисленные далее действия.

1. Откройте командную оболочку и выберите каталог, который будет содержать файлы базы данных.
2. Отыщите файл `derbyrun.jar`. В одних версиях JDK он может находиться в каталоге `jdk/db/lib`, а в других — в отдельном инсталляционном каталоге JavaDB. Мы будем обозначать каталог, содержащий `lib/derbyrun.jar`, как `derby`.
3. Выполните команду `java -jar derby/lib/derbyrun.jar server start`.
4. Удостоверьтесь еще раз, работает ли база данных должным образом. Создайте файл `ij.properties` с такими строками:

   ```
   ij.driver=org.apache.derby.jdbc.ClientDriver
   ij.protocol=jdbc:derby://localhost:1527/
   ij.database=COREJAVA;create=true
   ```

   В другой командной оболочке запустите интерактивный инструмент написания сценариев Derby (он называется `ij`), выполнив такую команду:

   ```
   java -jar derby/lib/derbyrun.jar ij -p ij.properties
   ```

   Теперь можно выполнять команды SQL, например:

   ```
   CREATE TABLE Greetings (Message CHAR(20));
   INSERT INTO Greetings VALUES ('Hello, World!');
   SELECT * FROM Greetings;
   DROP TABLE Greetings;
   ```

   Обратите внимание, что каждая команда должна завершаться точкой с запятой. Чтобы выйти, введите команду `EXIT;`.
5. После того как вы завершите работу с базой данных, остановите работу сервера с помощью команды

   ```
   java -jar derby/lib/derbyrun.jar server shutdown
   ```

Если вы используете другую базу данных, вам необходимо найти в документации сведения о запуске и останове сервера базы данных, а также о том, как подключиться к нему и как выполнять команды SQL.

Регистрация класса драйвера

Некоторые JAR-файлы JDBC (например, драйвер Derby, входящий в состав Java SE 6) автоматически регистрируют класс драйвера. В этом случае вы можете пропустить шаг ручной регистрации, который рассматриваencz в настоящем разделе. JAR-файл может автоматически зарегистрировать класс драйвера, если он содержит файл `META-INF/services/java.sql.Driver`. Чтобы проверить это, достаточно просто распаковать JAR-файл драйвера.



Механизм регистрации использует малоизвестную часть спецификации JAR; см. `http://java.sun.com/javase/6/docs/technotes/guides/jar/jar.html#Service%20Provider`. Для драйвера, совместимого с JDBC4, автоматическая регистрация является обязательной.

Если JAR-файл драйвера не поддерживает автоматическую регистрацию, потребуется узнать имя классов драйвера JDBC, используемого вашим поставщиком. Обычными именами драйверов являются:

```
org.apache.derby.jdbc.ClientDriver
org.postgresql.Driver
```

Существуют два способа регистрации драйвера посредством `DriverManager`. Один способ заключается в том, чтобы загрузить класс драйвера в вашу программу на Java. Например:

```
Class.forName("org.postgresql.Driver"); // загружает класс драйвера
```

Этот оператор загружает класс драйвера, выполняя тем самым статический инициализатор, который осуществляет регистрацию драйвера.

Другой способ заключается в том, чтобы задать свойство `jdbc.drivers`. Вы можете определить свойство посредством аргумента командной строки, как показано ниже:

```
java -Djdbc.drivers=org.postgresql.Driver Имя_Программы
```

Или же ваше приложение может установить системное свойство посредством следующего вызова:

```
System.setProperty("jdbc.drivers", "org.postgresql.Driver");
```

При неободимости можно 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 Java-программе можно с помощью приведенного ниже 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 подпротоколу, указанному в URL базы данных.

Метод `getConnection()` возвращает объект `Connection`, который используется для выполнения SQL-команд.

Чтобы соединиться с базой данных, необходимо знать имя пользователя базы данных и пароль.



По умолчанию Derby допускает соединения под любым именем пользователя, при этом она не проверяет пароль. Для каждого отдельно взятого пользователя генерируется отдельная схема. Именем пользователя по умолчанию является `app`.

Чтобы протестировать программу, код которой приведен в листинге 4.1, нужно сделать все то, что мы описали. Эта программа загружает параметры соединения из файла `database.properties` и осуществляет соединение с базой данных. Файл `database.properties` поставляется вместе с примером кода, который содержит информацию о соединении для базы данных Derby. Если вы используете другую базу данных, нужно поместить в файл вашу информацию о соединении, связанную с базой данных. Ниже показан пример 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 тестовая программа выполняет следующие команды SQL:

```
CREATE TABLE Greetings (Message CHAR(20))
INSERT INTO Greetings VALUES ('Hello, World!')
SELECT * FROM Greetings
```

В результате должна отобразиться строка

```
Hello, World!
```

После этого необходимо удалить таблицу с помощью команды

```
DROP TABLE Greetings
```

Чтобы выполнить этот тест, запустите базу данных и программу, как показано ниже:

```
java -classpath .:driverJAR TestDB
```



Устранить проблемы, связанные с JDBC, можно посредством трассировки JDBC. Вызовите метод `DriverManager.setLogWriter`, чтобы послать сообщения трассировки в `PrintWriter`. Вывод трассировки содержит подробный список действий JDBC. Большинство реализаций драйвера JDBC предоставляют дополнительные механизмы трассировки. Например, для Derby необходимо добавить опцию `traceFile` в JDBC URL:
`jdbc:derby://localhost:1527/COREJAVA;create=true;traceFile=trace.out`

Листинг 4.1. Содержимое файла `TestDB.java`

```
1.  import java.sql.*;
2.  import java.io.*;
3.  import java.util.*;
4.
5.  /**
6.   * Эта программа проверяет правильность настройки
7.   * базы данных и JDBC-драйвера.
8.   * @version 1.01 2004-09-24
9.   * @author Cay Horstmann
10.  */
11. class TestDB
12. {
13.    public static void main (String args[])
14.    {
15.       try
16.       {
17.          runTest();
18.       }
19.       catch (SQLException ex)
20.       {
21.          for (Throwable t : ex)
22.             t.printStackTrace();
23.       }
24.       catch (IOException ex)
25.       {
26.          ex.printStackTrace();
27.       }
28.    }
29.
```

```
30.    /**
31.     * Тестирование, включающее создание таблицы, добавление
32.     * значения, отображения содержимого таблицы и удаление
33.     * таблицы.
34.     */
35.    public static void runTest() throws SQLException, IOException
36.    {
37.       Connection conn = getConnection();
38.       try
39.       {
40.          Statement stat = conn.createStatement();
41.
42.          stat.executeUpdate(
43.             "CREATE TABLE Greetings (Message CHAR(20))");
44.          stat.executeUpdate(
45.             "INSERT INTO Greetings VALUES ('Hello, World!')");
46.
47.          ResultSet result = stat.executeQuery(
48.             "SELECT * FROM Greetings");
49.          if (result.next());
50.             System.out.println(result.getString(1));
51.          result.close();
52.          stat.executeUpdate("DROP TABLE Greetings");
53.       }
54.       finally
55.       {
56.          conn.close();
57.       }
58.    }
59.
60.    /**
61.     * Установка соединения с использованием свойств,
62.     * заданных в файле database.properties.
63.     * @return Соединение с базой данных
64.     */
65.    public static Connection getConnection() throws SQLException, IOException
66.    {
67.       Properties props = new Properties();
68.       FileInputStream in = new
69.          FileInputStream("database.properties");
70.       props.load(in);
71.       in.close();
72.
73.       String drivers = props.getProperty("jdbc.drivers");
74.
75.       if (drivers != null) System.setProperty("jdbc.drivers", drivers);
76.
77.       String url = props.getProperty("jdbc.url");
78.       String username = props.getProperty("jdbc.username");
79.       String password = props.getProperty("jdbc.password");
80.
81.       return DriverManager.getConnection(url, username, password);
82.    }
83. }
```

API java.sql.DriverManager 1.1

- `static Connection getConnection(String url, String user, String password)`
  Устанавливает соединение с данной базой данных и возвращает объект `Connection`.

Выполнение операторов SQL

Для выполнения оператора SQL нужно создать объект `Statement`. Для этой цели используется объект `Connection`, который можно получить, вызвав метод `DriverManager.getConnection()`:

```
Statement stat = conn.createStatement();
```

Затем потребуется создать строку с требуемой SQL-командой:

```
String command = "UPDATE Books"
   + " SET Price = Price - 5.00"
   + " WHERE Title NOT LIKE '%Introduction%'";
```

Далее необходимо вызвать метод `executeUpdate()` класса `Statement`:

```
stat.executeUpdate(command);
```

Метод `executeUpdate` возвращает количество строк, полученных в результате выполнения SQL-команды, или ноль для команд, которые не возвращают количества строк. Так, в приведенном выше примере будет возвращено количество книг, цена которых снижена на 5 долларов.

Метод `executeUpdate()` может применяться для выполнения команд `INSERT`, `UPDATE` и `DELETE`, а также команд определения данных, в частности, `CREATE TABLE` и `DROP TABLE`. Но для выполнения оператора `SELECT` необходимо использовать другой метод, а именно — `executeQuery()`. Существует также универсальный метод `execute()`, который может применяться для выполнения произвольных операторов SQL, но он используется в основном для интерактивного создания запросов.

Если вы составляете запрос, вас, конечно же, интересуют результаты. Метод `executeQuery()` возвращает объект `ResultSet`, который можно использовать для построчного просмотра результатов:

```
ResultSet rs = stat.executeQuery("SELECT * FROM Books");
```

Для анализа результирующего набора применяется приведенный ниже цикл:

```
while (rs.next())
{
   обработка строки
}
```



Способ последовательной обработки строк в классе `ResultSet` организован несколько иначе, чем в интерфейсе `java.util.Iterator`. В классе `ResultSet` итератор располагается в начале работы *перед* первой строкой. Поэтому для перемещения его к первой строке нужно использовать метод `next()`. Кроме этого, в нем нет метода `hasNext`. Вам придется продолжать вызывать `next` до тех пор, пока не будет возвращено значение `false`.

Строки в результирующем наборе располагаются совершено произвольно. Если порядок следования важен, его необходимо установить с помощью конструкции `ORDER BY`.

При обработке отдельной строки нужно с помощью специальных методов получить содержимое каждого столбца:

```
String isbn = rs.getString(1);
float price = rs.getDouble("Price");
```

Для каждого *типа* данных языка Java предусмотрен отдельный метод извлечения информации, например, `getString()` и `getDouble()`. Для каждого из них реализованы два варианта, один из них предполагает числовой, а второй — строковый параметр. При использовании числового параметра метод извлечет данные из столбца с указанным номером. Например, метод `rs.getString(1)` возвратит значение из первого столбца текущей строки.



В отличие от массивов, нумерация столбцов базы данных начинается с 1.

При использовании строкового параметра метод извлечет данные из столбца с указанным именем. Например, `rs.getDouble("Price")` возвратит значение из столбца с именем `Price`. Первый способ на основе числового параметра более эффективен, но строковые параметры улучшают восприятие и упрощают сопровождение кода.

Если указанный тип не соответствует фактическому типу, метод извлечения данных выполняет преобразование. Например, `rs.getString("Price")` преобразует число с плавающей точкой из столбца `Price` в строку.

API java.sql.Connection 1.1

- `Statement createStatement()`

  Создает объект `Statement`, который может использоваться для выполнения SQL-команд без параметров.

- `void close()`

  Закрывает текущее соединение и созданные в его рамках JDBC-ресурсы.

API java.sql.Statement 1.1

- `ResultSet executeQuery(String sqlQuery)`

  Выполняет заданную в виде строки SQL-команду и возвращает объект `ResultSet` с результатами выполнения этой команды.

- `int executeUpdate(String sqlStatement)`

  Выполняет заданные в виде строки SQL-команды `INSERT`, `UPDATE` и `DELETE`, а также команды определения данных `CREATE TABLE` и `DROP TABLE`. Он возвращает количество записей, задействованных в ходе выполнения этой команды, или `-1`, если для данной SQL-команды не создан счетчик обновлений.

- `boolean execute(String sqlStatement)`

  Выполняет заданную в виде строки SQL-команду и возвращает `true`, если эта команда предоставляет результирующий набор, или `false` в противном случае. Может создать множество результирующих наборов и обновления подсчетов. Для доступа к данным, полученным в результате выполнения команды, необходимо использовать метод `getResultSet()` или `getUpdateCount()`. Вопросы обработки множественных результатов рассматриваются в разделе “Множественные результаты”.

- `ResultSet getResultSet()`

  Возвращает объект `ResultSet` с результатами выполнения предыдущей команды или `null`, если выполнение предыдущей команды не дало никаких результатов. Его следует вызывать только один раз для каждой выполняемой SQL-команды.

- `int getUpdateCount()`

  Возвращает количество строк, задействованных в ходе выполнения предыдущей команды обновления, или `-1`, если для данной SQL-команды не создан счетчик обновлений. Его следует вызывать только один раз для каждой выполняемой SQL-команды.

- `void close()`

  Закрывает текущий объект `Statement` и связанные с ним результирующие наборы.

- `boolean isClosed()` 6

  Возвращает значение `true` при закрытии этой команды.

API **`java.sql.ResultSet`**

- `boolean next()`

  Перемещает указатель текущей строки в результирующем наборе на одну позицию вперед. После прохождения последней строки возвращает `false`. Учтите, что данный метод нужно обязательно вызвать для перемещения указателя к первой строке.

- *`Xxx`* `get`*`Xxx`*`(int columNumber)`
- *`Xxx`* `get`*`Xxx`*`(String columLabel)`

  (*`Xxx`* обозначает тип данных, например, `int`, `double`, `String`, `Date` и т.д.) Возвращает значение столбца, заданного посредством номера или метки, с преобразованием к указанному типу данных. Учтите, что допускаются не все варианты преобразования типов. Метка столбца — это метка, определенная в SQL-конструкции `AS`, или имя столбца, если конструкция `AS` не используется.

- `int findColumn(String columnName)`

  Возвращает номер столбца с заданным именем.

- `void close()`

  Закрывает текущий результирующий набор.

- `boolean isClosed()` 6

  Возвращает значение `true` при закрытии этой команды.

Управление соединениями, командами и результирующими наборами

Каждый объект `Connection` может создать один или несколько объектов `Statement`. Один и тот же объект `Statement` можно использовать для нескольких не связанных между собой команд и запросов. Однако для такого объекта допускается наличие не более одного открытого результирующего набора. Если требуется выполнить несколько команд с одновременным анализом предоставленных ими результатов, вам понадобится несколько объектов `Statement`.

Не следует, однако, забывать, что, по крайней мере, одна широко используемая база данных (Microsoft SQL Server) взаимодействует с JDBC-драйвером, кото-

рый допускает работу только с одним объектом Statement. Количество объектов Statement, одновременно поддерживаемых JDBC-драйвером, можно узнать, вызвав метод getMaxStatements() класса DatabaseMetaData.

Подобное ограничение кажется излишним, но на практике оно почти никогда не влияет на качество работы, так как необходимость одновременно использовать несколько результирующих наборов возникает крайне редко. Если же результаты связаны друг с другом, вы всегда можете составить сложный запрос и анализировать один набор. Гораздо выгоднее предоставить базе данных возможность самостоятельно соединить результаты, чем перебирать различные наборы в Java-программе.

Закончив работу с классом ResultSet, Statement или Connection, следует как можно скорее вызвать метод close(). Дело в том, что перечисленные объекты используют структуры данных большого объема и вам вряд ли захочется ожидать, пока процедура сборки мусора освободит ресурсы.

Метод close() класса Statement автоматически закрывает связанные с ним результирующие наборы (если, конечно, эти наборы открыты). Аналогично метод close() класса Connection закрывает все объекты Statement для этого соединения.

Если время существования соединения невелико, нет необходимости заботиться о закрытии объектов Statement и результирующих наборов. Нужно лишь поместить вызов метода close() в блок finally; этим вы гарантируете, что объект соединения не останется открытым.

```
try
{
   Connection conn = . . .;
   try
   {
      Statement stat = conn.createStatement();
      ResultSet result = stat.executeQuery(queryString);
      обработка результатов запроса
   }
   finally
   {
      conn.close();
   }
}
catch (SQLException ex)
{
   обработка исключения
}
```



Желательно использовать блок try/finally для закрытия соединения и отдельный блок try/catch для обработки исключений. Разделив блоки try различного назначения, вы сделаете свою программу более удобной для чтения и сопровождения.

Анализ SQL-исключений

Каждое SQL-исключение имеет цепочку объектов SQLException, которые извлекаются посредством метода getNextException. Эта цепочка исключений является дополнением цепочки "cause" объектов Throwable, имеющихся в каждом исключении. (Подробные сведения об исключениях Java можно найти в главе 11 первого тома.) Чтобы полностью перечислить все исключения, может потребоваться два вложенных цикла. К счастью, в Java SE 6 был усовершенствован класс SQLException для реализации интерфейса Iterable<Throwable>.

Метод `iterator()` дает `Iterator<Throwable>`, который осуществляет перебор в обеих цепочках, сначала проходя по цепочке "cause" первого `SQLException`, а затем переходя к следующему `SQLException` и т.д. Вы можете просто использовать улучшенный цикл `for`:

```
for (Throwable t : sqlException)
{
   любые действия с t
}
```

Чтобы продолжить дальнейший анализ `SQLException`, можно вызвать методы `getSQLState` и `getErrorCode` в `SQLException`. Первый метод дает строку, стандартизированную в соответствии с X/Open или SQL:2003. (Вызовите метод `DatabaseMetadata getSQLStateType`, чтобы узнать, какой стандарт используется вашим драйвером.) Что же касается кода ошибки, то у разных поставщиков он разный.

В версии Java SE 6 исключения SQL организованы в виде дерева наследования (оно показано на рис. 4.6). Благодаря ему, вы сможете перехватывать определенные типы ошибок, независимо от предпочтений поставщиков.

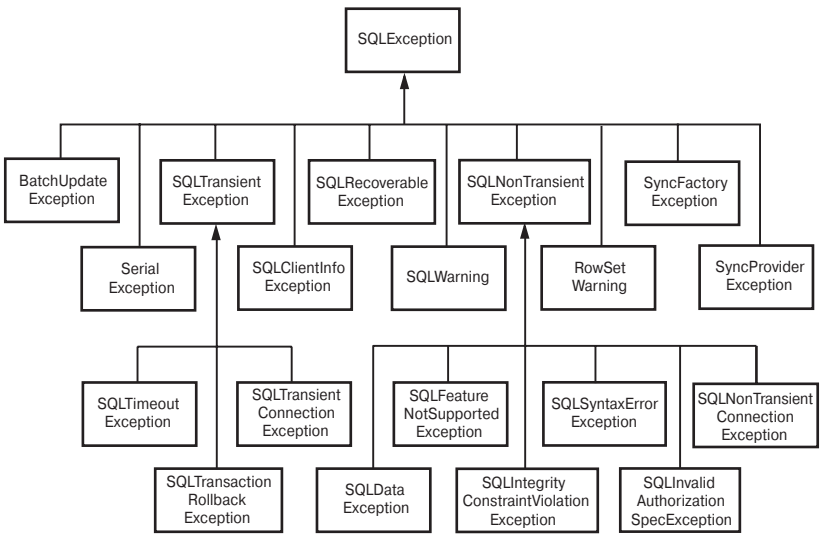


Рис. 4.6. Типы исключений SQL

Кроме того, драйвер базы данных может сообщать о некритичных ситуациях в форме предупреждений. Вы можете получать предупреждения от соединений, команд и результирующих наборов. Класс `SQLWarning` является подклассом `SQLException` (даже если `SQLWarning` не генерируется в виде исключения). Посредством вызова методов `getSQLState` и `getErrorCode` можно получить дальнейшую информацию о предупреждениях. Подобно исключениям SQL, предупреждения ставятся в цепочку. Чтобы получить все сообщения, потребуется следующий цикл:

```
SQLWarning w = stat.getWarning();
while (w != null)
{
   любые действия с w
   w = w.nextWarning();
}
```

Подкласс DataTruncation класса SQLWarning используется в тех случаях, когда данные считываются из базы и в этот момент происходит неожиданное их усечение. Если усечение данных произошло в команде обновления, то DataTruncation генерируется в форме исключения.

API java.sql.Warning

- SQLWarning getNextWarning()

  Возвращает предупреждение, следующее за текущим предупреждением, или null, если будет достигнут конец цепочки.

API Java.sql.Connection

API java.sql.Statement

API java.sql.ResultSet

- SQLWarning getWarnings()
- SQLWarning getWarnings()

  Возвращает первое сообщение из очереди, или null, если в очереди нет предупреждений.

API java.sql.DataTruncation

- boolean getParameter()

  Возвращает значение true, если усечение данных применяется к параметру, и значение false, если оно применяется к столбцу.

- int getIndex()

  Возвращает индекс усеченного параметра или столбца.

- int getDatabaseSize()

  Возвращает количество байт, которые необходимо передать, или -1, если значение неизвестно.

- int GetTransferSize()

  Возвращает количество байт, которые были на самом деле перемещены, или -1, если значение неизвестно.

Заполнение базы данных

Попробуем теперь создать первую программу на основе JDBC. Было бы просто замечательно, если бы при этом мы могли выполнить некоторые из перечисленных выше запросов. Но, к сожалению, это невозможно, потому что база данных пуста. Нам нужно заполнить базу данных, и сделать это можно без особого труда: посредством набора инструкций SQL для создания таблиц и вставки в них данных. Большинство СУБД могут обрабатывать набор инструкций SQL из текстового файла, однако в этом случае в игру вступают досадные различия в завершающих символах операторов и другие проблемы, связанные с синтаксисом.

В связи с этим, мы воспользуемся JDBC для того, чтобы создать простую программу построчного чтения файла с такими инструкциями SQL и их выполнения.

Ниже показано, как выглядит программа чтения текстового файла с данными:

```
CREATE TABLE Publisher (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 код программы `ExecSQL`, которая считывает текстовый файл с SQL-командами и выполняет их. В этом случае не важно, что вы производите чтение посредством кода; мы лишь показали программу, чтобы вы могли самостоятельно заполнить базу данных и выполнить примеры в оставшейся части этой главы.

Убедитесь в том, что сервер базы данных работает, и запустите программу, сделав следующие вызовы:

```
java -classpath .:путь_к_драйверу ExecSQL Books.sql
java -classpath .:путь_к_драйверу ExecSQL Authors.sql
java -classpath .:путь_к_драйверу ExecSQL Publishers.sql
java -classpath .:путь_к_драйверу ExecSQL BooksAuthors.sql
```

Перед запуском этой программы проверьте содержимое файла свойств `database.properties` и удостоверьтесь, что оно соответствует вашей среде (см. раздел "Соединение с базой данных" ранее в главе).



Ваша база данных может иметь утилиту для прямого чтения файлов SQL. Например, в базе данных Derby вы можете выполнить такую команду:

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties Books.sql
```

(Файл `ij.properties` описан в разделе "Запуск базы данных".)

Как вариант, если вы знакомы с Ant, вы можете использовать задачу `sql`.

В формате данных для команды `ExecSQL` мы допускаем вставку точки с запятой в конце каждой строки, так как большинство утилит баз данных, к которым причисляется и Ant, используют этот формат.

Ниже перечислены основные этапы выполнения программы `ExecSQL`.

1. Устанавливается соединение с базой данных. Метод `getConnection()` считывает свойства из файла `database.properties` и добавляет `jdbc.drivers` в список системных свойств. Диспетчер драйверов использует свойство `jdbc.drivers` для загрузки соответствующего драйвера базы данных. Для установления соединения метод `getConnection()` применяет свойства `jdbc.url`, `jdbc.username` и `jdbc.password`.
2. Открывается файл с SQL-командами. Если такого файла нет, пользователю предлагается ввести команды с консоли.
3. Все заданные команды выполняются с помощью универсального метода `execute()`. При получении результирующего набора этот метод возвращает значение `true`. Во всех четырех файлах с SQL-командами в конце содержится команда `SELECT *`, поэтому вы сможете просмотреть все успешно вставленные записи.
4. Полученные результаты отображаются на экране. Поскольку наборы универсальны, необходимо использовать *метаданные*, которые позволят определить, сколько столбцов содержится в результирующем наборе. Более подробно метаданные рассматриваются далее в главе.

5. При наличии какого-либо исключения, связанного с выполнением SQL-команды, выводятся сведения, находящиеся в объекте исключения и во всех исключениях, принадлежащих цепочке.

В листинге 4.2 приведен код этой программы.

Листинг 4.2. Содержимое файла `ExecSQL.java`

```
1.  import java.io.*;
2.  import java.util.*;
3.  import java.sql.*;
4.  /**
5.   * Программа, демонстрирующая выполнение SQL-команд,
6.   * которые содержатся в текстовом файле.
7.   * Вызывается следующим образом:
8.   * java -classpath путь_к_драйверу:. ExecSQL файл
9.   * @version 1.30 2004-08-05
10.  * @author Cay Horstmann
11.  */
12. class ExecSQL
13. {
14.    public static void main (String args[])
15.    {
16.       try
17.       {
18.          Scanner in;
19.          if (args.length == 0) in = new Scanner(System.in);
20.          else in = new Scanner(new File(args[0]));
21.
22.          Connection conn = getConnection();
23.          try
24.          {
25.             Statement stat = conn.createStatement();
26.             while (true)
27.             {
28.                if (args.length == 0) System.out.println("Enter command or EXIT to exit:");
29.
30.                if (!in.hasNextLine()) return;
31.
32.                String line = in.nextLine();
33.                if (line.equalsIgnoreCase("EXIT")) return;
34.                if (line.trim().endsWith(";")) // удаление завершающей точки с запятой
35.                {
36.                   line = line.trim();
37.                   line = line.substring(0, line.length() - 1);
38.                }
39.                try
40.                {
41.                   boolean hasResultSet = stat.execute(line);
42.                   if (hasResultSet) showResultSet(stat);
43.                }
44.                catch (SQLException e)
45.                {
46.                   for (Throwable e : ex)
47.                      e.printStackTrace();
48.                }
49.             }
50.          }
```

```
51.         finally
52.         {
53.            conn.close();
54.         }
55.       }
56.       catch (SQLException e)
57.       {
58.         for (Throwable t : e)
59.         t.printStackTrace();
60.       }
61.       catch (IOException e)
62.       {
63.         e.printStackTrace();
64.       }
65.     }
66.
67.     /**
68.      * Установление соединения на основе свойств,
69.      * заданных в файле database.properties.
70.      * @return Соединение с базой данных
71.      */
72.     public static Connection getConnection() throws SQLException, IOException
73.     {
74.       Properties props = new Properties();
75.       FileInputStream in = new
76.         FileInputStream("database.properties");
77.       props.load(in);
78.       in.close();
79.
80.       String drivers = props.getProperty("jdbc.drivers");
81.       if (drivers != null)
82.         System.setProperty("jdbc.drivers", drivers);
83.
84.       String url = props.getProperty("jdbc.url");
85.       String username = props.getProperty("jdbc.username");
86.       String password = props.getProperty("jdbc.password");
87.
88.       return DriverManager.getConnection(url, username, password);
89.     }
90.
91.     /**
92.      * Вывод результирующего набора.
93.      * @param Объект Statement, результаты которого должны
94.      * быть выведены
95.      */
96.     public static void showResultSet(Statement stat) throws SQLException
97.     {
98.       ResultSet result = stat.getResultSet();
99.       ResultSetMetaData metaData = result.getMetaData();
100.      int columnCount = metaData.getColumnCount();
101.
102.      for (int i = 1; i <= columnCount; i++)
103.      {
104.        if (i > 1) System.out.print(", ");
105.        System.out.print(metaData.getColumnLabel(i));
106.      }
107.      System.out.println();
```

```
108.      while (result.next())
109.      {
110.         for (int i = 1; i <= columnCount; i++)
111.         {
112.            if (i > 1) System.out.print(", ");
113.            System.out.print(result.getString(i));
114.         }
115.         System.out.println();
116.      }
117.      result.close();
118.   }
119. }
```

Выполнение запроса

В данном разделе рассматривается процедура создания приложения, которое сможет выполнять запросы к базе данных COREJAVA. Для работы этой программы в базе данных нужно создать описанные в предыдущем разделе таблицы. На рис. 4.7 показан внешний вид окна приложения.

В этой программе можно выбрать имя автора и/или издательство (либо оставить предлагаемый по умолчанию выбор всех авторов и издательств — Any), а затем щелкнуть на кнопке Query (Запрос), после чего в текстовой области будут отображены все книги, которые соответствуют заданным условиям выбора.

С помощью программы можно изменить информацию в базе данных. Выберите нужное издательство и введите число, обозначающее изменение цены, в поле редактирования возле кнопки изменения цены Change price (Изменить цену). После щелчка на этой кнопке цены на все книги этого издательства будут изменены на указанную величину, а в текстовой области в нижней части окна вы увидите сообщение о том, сколько записей было изменено. Для предотвращения случайного изменения сразу всех цен здесь не допускается выбор сразу всех книг. При изменении цен выбор автора игнорируется. После изменения цены можно выполнить запрос для проверки измененных данных.

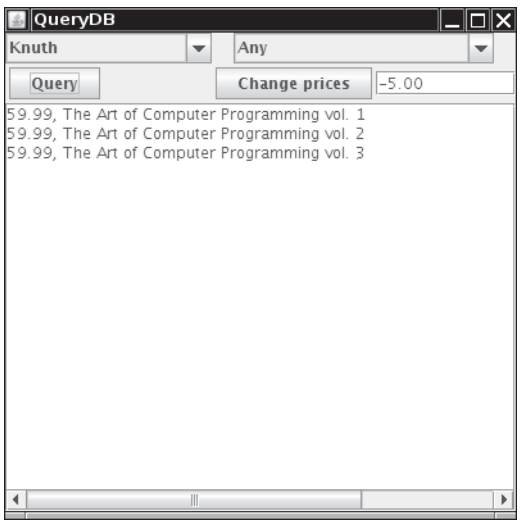


Рис. 4.7. Приложение QueryDB

Предварительно подготовленные команды

В данной программе мы используем новое средство — *предварительно подготовленные* (prepared) операторы SQL . Рассмотрим запрос на выбор всех книг некоторого издательства, независимо от их авторов:

```
SELECT Books.Price, Books.Title
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
AND Publishers.Name = имя_издательства_из_списка
```

Вместо создания отдельной SQL-команды для каждого запроса со стороны пользователя следует *предварительно подготовить* запрос с подстановочной переменной и многократно использовать его, меняя только значение переменной. Эта возможность существенно повышает эффективность работы программы. Перед каждым выполнением запроса СУБД создает план его эффективного исполнения. Предварительно подготавливая запрос для последующего многократного повторного использования, можно избежать повторного создания плана.

Подстановочная переменная в запросе обозначается символом `?`. При использовании нескольких подстановочных переменных нужно внимательно следить за их расположением, чтобы правильно выполнять подстановку фактических значений. Ниже показано, как выглядит предварительно подготовленный запрос в коде программы:

```
String publisherQuery =
   "SELECT Books.Price, Books.Title" +
   " FROM Books, Publishers" +
   " WHERE Books.Publisher_Id =
      Publishers.Publisher_Id AND Publishers.Name = ?";
PreparedStatement publisherQueryStat =
   conn.prepareStatement(publisherQuery);
```

Перед выполнением предварительно подготовленного запроса нужно с помощью метода `set()` связать подстановочные переменные с их фактическими значениями. Как и при использовании методов `get()` класса `ResultSet`, для разных типов данных предусмотрены разные методы `set()`. Вот как задается строковое значение с названием издательства:

```
publisherQueryStmt.setString(1, publisher);
```

Первый параметр обозначает номер позиции подстановочной переменной, а второй — ее фактическое значение.

При повторном использовании предварительно подготовленного запроса с несколькими подстановочными переменными все связи остаются в силе, если только они не изменены с помощью методов `set()` или `clearParameters()`. Это значит, что методы `set()` нужно вызывать только для тех подстановочных переменных, которые изменяются в последующих запросах.

После установления связи между подстановочными переменными и их фактическими значениями можно приступать к выполнению запроса:

```
ResultSet rs = publisherQueryStat.executeQuery();
```



Создание запроса вручную, посредством конкатенации строк, представляет собой трудоемкое и потенциально опасное занятие. Вам придется заботиться о специальных символах (например, кавычках), и, если запрос включает данные, вводимые пользователем, вам придется быть готовыми к ошибкам со стороны пользователей. Таким образом, предварительно подготовленные запросы гораздо удобнее.

Обновление цены осуществляется посредством оператора UPDATE. Обратите внимание, что для этого используется метод executeUpdate(), а не executeQuery(). Дело в том, что UPDATE не возвращает результирующий набор, который в этом случае не нужен. Метод executeUpdate() возвращает лишь счетчик обновленных записей, который мы и отображаем в текстовой области:

```
int r = priceUpdateStmt.executeUpdate();
result.setText(r + " records updated");
```



Объект PreparedStatement становится недействительным после того, как соответствующий объект Connection закрывается. Однако многие драйверы баз данных автоматически кэшируют предварительно подготовленные запросы. Если один и тот же запрос подготавливается дважды, СУБД лишь повторно использует имеющиеся сведения. Поэтому, вызывая метод prepareStatement(), не следует беспокоиться о накладных расходах.

Ниже описаны действия, выполняемые в процессе работы программы.

- Заполнение текстовых окон имен авторов и названий издательств осуществляется посредством выполнения двух запросов.
- Слушатель кнопки Query проверяет, какой тип запроса требуется. При первом выполнении предварительного запроса выбранного типа для него создается оператор SQL, в котором подстановочная переменная имеет значение null. После этого устанавливается связь между подстановочными переменными и их фактическими значениями, а затем выполняется запрос.
- Запросы с указанием имени автора имеют более сложную структуру. Поскольку одна книга может иметь несколько авторов, в таблице BooksAuthors задается соответствие между авторами и книгами. Предположим, что книга с ISBN 0-201-96426-0 имеет двух авторов с кодами DATE и DARW. В таком случае таблица BooksAuthors будет содержать следующие строки:

  ```
  0-201-96426-0, DATE, 1
  0-201-96426-0, DARW, 2
  ```

  В третьем столбце указан порядковый номер авторов. (Для этого нельзя использовать информацию о расположении строк в таблице, поскольку в реляционной базе данных порядок следования записей не фиксирован.) Запрос должен соединить таблицы Books, BooksAuthors и Authors, а затем сравнить с полученными значениями указанное пользователем имя автора.

  ```
  SELECT Books.Price, Books.Title 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 = ?
  ```



Некоторые программисты Java стараются избегать создания сложных операторов SQL. Как ни странно, но они склонны делать то же самое с помощью Java-кода, причем при этом приходится обрабатывать несколько результирующих наборов. Учтите, что СУБД выполняет запросы *гораздо* эффективнее, чем Java-программа, поскольку СУБД оптимизирована именно для этой цели. Помните основное правило: то, что можно сделать средствами SQL, не следует делать средствами Java.

- Слушатель кнопки Change prices (Изменить цены) выполняет оператор UPDATE. Следует отметить, что этот запрос имеет довольно сложную структуру, поскольку конструкция WHERE оператора UPDATE должна получить *код* издательства по известному *названию* издательства. Эта проблема решается с помощью вложенного запроса, например, так, как показано ниже:

```
UPDATE Books
SET Price = Price + ?
 WHERE Books.Publisher_Id =
   (SELECT Publisher_Id FROM Publishers WHERE Name = ?)
```

- Инициализация соединения и объектов Statement осуществляется в конструкторе. Эти объекты используются в течение всего жизненного цикла программы. Непосредственно перед окончанием работы программы следует перехватить событие закрытия окна и закрыть все объекты.

Полностью код этой программы представлен в листинге 4.3.

Листинг 4.3. Содержимое файла QueryDB.java

```
1.    import java.sql.*;
2.    import java.awt.*;
3.    import java.awt.event.*;
4.    import java.io.*;
5.    import java.util.*;
6.    import javax.swing.*;
7.
8.    /**
9.     * Программа, демонстрирующая использование сложных запросов.
10.   * @version 1.23 2007-06-28
11.   * @author Cay Horstmann
12.   */
13.  public class QueryDB
14.  {
15.     public static void main(String[] args)
16.     {
17.      EventQueue.invokeLater(new Runnable()
18.         {
19.            public void run()
20.            {
21.               JFrame frame = new QueryDBFrame();
22.                    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.               frame.setVisible(true);
24.            }
25.         });
26.   }
27. }
28.
29. /**
30.  * Этот фрейм содержит раскрывающиеся списки для ввода
31.  * параметров запроса, текстовую область для отображения
32.  * результатов запроса, а также кнопки для выполнения
33.  * запроса и обновления данных.
34.  */
35. class QueryDBFrame extends JFrame
36. {
37.    public QueryDBFrame()
38.    {
```

```
39.       setTitle("QueryDB");
40.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
41.       setLayout(new GridBagLayout());
42.
43.       authors = new JComboBox();
44.       authors.setEditable(false);
45.       authors.addItem("Any");
46.
47.       publishers = new JComboBox();
48.       publishers.setEditable(false);
49.       publishers.addItem("Any");
50.
51.       result = new JTextArea(4, 50);
52.       result.setEditable(false);
53.
54.       priceChange = new JTextField(8);
55.       priceChange.setText("-5.00");
56.       try
57.       {
58.          conn = getConnection();
59.          Statement stat = conn.createStatement();
60.          String query = "SELECT Name FROM Authors";
61.          ResultSet rs = stat.executeQuery(query);
62.          while (rs.next())
63.             authors.addItem(rs.getString(1));
64.          rs.close();
65.
66.          query = "SELECT Name FROM Publishers";
67.          rs = stat.executeQuery(query);
68.          while (rs.next())
69.             publishers.addItem(rs.getString(1));
70.          rs.close();
71.          stat.close();
72.       }
73.       catch (SQLException e)
74.       {
75.          for (Throwable t : e)
76.             result.append(t.getMessage());
77.       }
78.       catch (IOException e)
79.       {
80.          result.setText("" + e);
81.       }
82.
83.       // Мы используем класс GBC (см. главу 9 первого тома).
84.       add(authors, new GBC(0, 0, 2, 1));
85.
86.       add(publishers, new GBC(2, 0, 2, 1));
87.
88.       JButton queryButton = new JButton("Query");
89.       queryButton.addActionListener(new ActionListener()
90.          {
91.             public void actionPerformed(ActionEvent event)
92.             {
93.                executeQuery();
94.             }
95.          });
```

```
96.       add(queryButton, new GBC(0, 1, 1, 1).setInsets(3));
97.       JButton changeButton = new JButton("Change prices");
98.       changeButton.addActionListener(new ActionListener()
99.          {
100.            public void actionPerformed(ActionEvent event)
101.            {
102.               changePrices();
103.            }
104.         });
105.      add(changeButton, new GBC(2, 1, 1, 1).setInsets(3));
106.
107.      add(priceChange,
108.         new GBC(3, 1, 1, 1).setFill(GBC.HORIZONTAL));
109.
110.      add(new JScrollPane(result), new
111.         GBC(0, 2, 4, 1).setFill(GBC.BOTH).setWeight(100, 100));
112.
113.      addWindowListener(new WindowAdapter()
114.         {
115.            public void windowClosing(WindowEvent event)
116.            {
117.               try
118.               {
119.                  if (conn != null) conn.close();
120.               }
121.               catch (SQLException e)
122.               {
123.                  for (Throwable t : e)
124.                     t.printStackTrace();
125.               }
126.            }
127.         });
128.   }
129.   /**
130.    * Выполнение запроса.
131.    */
132.   private void executeQuery()
133.   {
134.      ResultSet rs = null;
135.      try
136.      {
137.         String author = (String) authors.getSelectedItem();
138.         String publisher = (String) publishers.getSelectedItem();
139.         if (!author.equals("Any") && !publisher.equals("Any"))
140.         {
141.            if (authorPublisherQueryStmt == null)
142.         authorPublisherQueryStmt = conn.prepareStatement(authorPublisherQuery);
143.            authorPublisherQueryStmt.setString(1, author);
144.            authorPublisherQueryStmt.setString(2, publisher);
145.            rs = authorPublisherQueryStmt.executeQuery();
146.         }
147.         else if (!author.equals("Any") &&
148.                  publisher.equals("Any"))
149.         {
150.            if (authorQueryStmt == null)
151.              authorQueryStmt = conn.prepareStatement(authorQuery);
152.            authorQueryStmt.setString(1, author);
```

```
153.          rs = authorQueryStmt.executeQuery();
154.        }
155.        else if (author.equals("Any") &&
156.               !publisher.equals("Any"))
157.        {
158.          if (publisherQueryStmt == null)
159.            publisherQueryStmt = conn.prepareStatement(publisherQuery);
160.          publisherQueryStmt.setString(1, publisher);
161.          rs = publisherQueryStmt.executeQuery();
162.        }
163.        else
164.        {
165.          if (allQueryStmt == null)
166.            allQueryStmt = conn.prepareStatement(allQuery);
167.          rs = allQueryStmt.executeQuery();
168.        }
169.        result.setText("");
170.        while (rs.next())
171.        {
172.          result.append(rs.getString(1));
173.          result.append(", ");
174.          result.append(rs.getString(2));
175.          result.append("\n");
176.        }
177.        rs.close();
178.      }
179.      catch (SQLException e)
180.      {
181.        for (Throwable t : e)
182.          result.append(t.getMessage());
183.      }
184.    }
185.    /**
186.     * Обновление данных для изменения цены.
187.     */
188.    public void changePrices()
189.    {
190.      String publisher = (String) publishers.getSelectedItem();
191.      if (publisher.equals("Any"))
192.      {
193.        result.setText("I am sorry, but I cannot do that.");
194.        return;
195.      }
196.      try
197.      {
198.        if (priceUpdateStmt == null)
199.          priceUpdateStmt = conn.prepareStatement(priceUpdate);
200.        priceUpdateStmt.setString(1, priceChange.getText());
201.        priceUpdateStmt.setString(2, publisher);
202.        int r = priceUpdateStmt.executeUpdate();
203.        result.setText(r + " records updated.");
204.      }
205.      catch (SQLException e)
206.      {
207.        for (Throwable t : e)
208.          result.append(t.getMessage());
209.      }
210.    }
```

```
211.   /**
212.    * Установка соединения с использованием свойств,
213.    * заданных в файле database.properties.
214.    * @return Соединение с базой данных
215.    */
216.   public static Connection getConnection() throws SQLException, IOException
217.   {
218.      Properties props = new Properties();
219.      FileInputStream in = new
220.         FileInputStream("database.properties");
221.      props.load(in);
222.      in.close();
223.
224.      String drivers = props.getProperty("jdbc.drivers");
225.      if (drivers != null)
226.         System.setProperty("jdbc.drivers", drivers);
227.      String url = props.getProperty("jdbc.url");
228.      String username = props.getProperty("jdbc.username");
229.      String password = props.getProperty("jdbc.password");
230.
231.      return DriverManager.getConnection(url, username, password);
232.   }
233.
234.   public static final int DEFAULT_WIDTH = 400;
235.   public static final int DEFAULT_HEIGHT = 400;
236.
237.   private JComboBox authors;
238.   private JComboBox publishers;
239.   private JTextField priceChange;
240.   private JTextArea result;
241.   private Connection conn;
242.   private PreparedStatement authorQueryStmt;
243.   private PreparedStatement authorPublisherQueryStmt;
244.   private PreparedStatement publisherQueryStmt;
245.   private PreparedStatement allQueryStmt;
246.   private PreparedStatement priceUpdateStmt;
247.
248.   private static final String authorPublisherQuery =
249.      "SELECT Books.Price, Books.Title" +
250.      " FROM Books, BooksAuthors, Authors, Publishers" +
251.      " WHERE Authors.Author_Id = BooksAuthors.Author_Id" +
252.      " AND BooksAuthors.ISBN = Books.ISBN" +
253.      " AND Books.Publisher_Id = Publishers.Publisher_Id" +
254.      " AND Authors.Name = ?" +
255.      " AND Publishers.Name = ?";
256.
257.   private static final String authorQuery =
258.      "SELECT Books.Price, Books.Title" +
259.      " FROM Books, BooksAuthors, Authors" +
260.      " WHERE Authors.Author_Id = BooksAuthors.Author_Id" +
261.      " AND BooksAuthors.ISBN = Books.ISBN" +
262.      " AND Authors.Name = ?";
263.
264.   private static final String publisherQuery =
265.      "SELECT Books.Price, Books.Title FROM Books, Publishers" +
266.      " WHERE Books.Publisher_Id = Publishers.Publisher_Id" +
267.      " AND Publishers.Name = ?";
```

```
268.   private static final String allQuery =
269.         "SELECT Books.Price, Books.Title FROM Books";
270.
271.   private static final String priceUpdate =
272.      "UPDATE Books " + "SET Price = Price + ? " +
273.      " WHERE Books.Publisher_Id =" +
274.      " (SELECT Publisher_Id FROM Publishers WHERE Name = ?)";
275.  }
276. }
```

API java.sql.Connection 1.1

- `PreparedStatement prepareStatement(String sql)`

  Возвращает объект `PreparedStatement`, содержащий предварительно созданную команду. Строка `sql` содержит SQL-команду с одной или несколькими подстановочными переменными, обозначенными вопросительными знаками.

API java.sql.PreparedStatement 1.1

- `void set`*`Xxx`*`(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 предварительно подготовленном запросе.

- `ResultSet executeQuery()`

  Выполняет предварительно подготовленный запрос и возвращает объект `ResultSet`.

- `int executeUpdate()`

  Выполняет предварительно подготовленные SQL-команды `INSERT`, `UPDATE` или `DELETE`, представленные в объекте `PreparedStatement`. Возвращает количество задействованных строк или значения `0` для операторов определения данных вроде `CREATE TABLE`.

Чтение и запись больших объектов

В дополнение к числам, строкам и датам, многие базы данных могут сохранять *большие объекты* (large object — LOB), к числу которых относятся изображения и другие данные. В языке SQL существует понятие больших двоичных объектов (binary large object — BLOB) и больших символьных объектов (character large object — CLOB).

Чтобы прочитать LOB, нужно сначала выполнить оператор SQL, а затем вызвать метод `getBlob` или `getClob` в `ResultSet`. В результате вы получите объект типа `Blob` или `Clob`. Чтобы получить двоичные данные из `Blob`, вызовите метод `getBytes` или `getInputStream`. Например, если у вас имеется таблица с изображениями на книжных обложках, то 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.getInputStream());
}
```

Подобным образом, если вы извлечете объект `Clob`, вы можете получить символьные данные посредством вызова метода `String` или `getCharacterStream`.

Чтобы поместить LOB в базу данных, нужно вызвать метод `createLOB` или `createClob` на вашем объекте `Connection`, получить выходной поток или писатель для LOB, записать данные и сохранить объект в базе данных. Например, ниже показан пример того, как можно сохранить изображение:

```
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();
```

API 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

  Получает BLOB или CLOB в данном столбце.

API java.sql.Blob 1.2

- `long length()`

  Получает длину данного BLOB.

- `byte[] getBytes(long startPosition, long length)`

  Получает данные в данном диапазоне из этого BLOB.

- `InputStream getBinaryStream()`
- `InputStream getBinaryStream(long startPosition, long length)`

  Возвращает поток для чтения данных в данном BLOB или данном диапазоне.

- `OutputStream setBinaryStream(long startPosition)` 1.4

  Возвращает выходной поток для записи в данный BLOB, начиная с данной позиции.

API java.sql.Clob 1.2

- `long length()`

  Получает количество символов в данном CLOB.

- `String getSubString(long startPosition, long length)`

  Получает символы в данном диапазоне из данного BLOB.

- `Reader getCharacterStream()`
- `Reader getCharacterStream(long startPosition, long length)`

  Возвращает читатель (а не поток) для чтения символов в данном CLOB или в данном диапазоне.
- `Writer setCharacterStream(long startPosition)` 1.4

  Возвращает писатель (а не поток) для записи в данный CLOB, начиная с данной позиции.

API **java.sql.Connection 1.1**

- `Blob createBlob()` 6
- `Clob createClob()` 6

  Создает пустой BLOB или CLOB.

Синтаксис перехода в SQL

Синтаксис "перехода" (escape syntax) поддерживает средства, которые обычно поддерживаются базами данных, но в разных вариантах, в зависимости от конкретного синтаксиса базы данных. Драйвер JDBC отвечает за преобразование синтаксиса перехода в синтаксис конкретной базы данных.

Переходы предусмотрены для следующих средств:

- литералы времени и даты;
- вызов скалярных функций;
- вызов хранимых процедур;
- внешние соединения;
- символ перехода в операциях `LIKE`.

Литералы времени и даты сильно отличаются от одной базы данных к другой. Чтобы вставить литерал времени или даты, нужно определить значение в формате ISO 8601 (`http://www.cl.cam.ac.uk/~mgk25/iso-time.html`). После этого драйвер преобразует литерал в родной формат. Для значений `DATE`, `TIME` или `TIMESTAMP` используются литералы `d`, `t` и `ts`:

```
{d '2008-01-24'}
{t '23:59:59'}
{ts '2008-01-24 23:59:59.999'}
```

Скалярной называется функция, возвращающая одно значение. В базах данных используется большое множество функций, но с разными именами. В спецификации JDBC указаны стандартные имена, преобразованные в имена, специфические для баз данных. Чтобы вызвать функцию, внедрите стандартное имя функции и аргументы, как показано ниже:

```
{fn left(?, 20)}
{fn user()}
```

В спецификации JDBC можно найти полный список поддерживаемых имен функций.

Хранимой процедурой (stored procedure) называется процедура, которая выполняется в базе данных, и которая написана на языке, специфичном для базы данных. Чтобы вызвать хранимую процедуру, используйте переход `call`. Если процедура не имеет параметров, то использовать скобки не нужно.

Чтобы сохранить возвращаемое значение, используйте символ =:

```
{call PROC1(?, ?)}
{call PROC2}
{call ? = PROC3(?)}
```

Внешнее соединение (outer join) двух таблиц не требует, чтобы строки в каждой таблице совпадали в соответствии с условием соединения. Например, запрос

```
SELECT * FROM {oj Books LEFT OUTER JOIN Publishers ON Books.Publisher_Id =
Publisher.Publisher_Id}
```

содержит книги, в которых `Publisher_Id` не имеет совпадений в таблице `Publishers`, со значениями `NULL`, показывающими, что совпадения отсутствуют. Вам может потребоваться `RIGHT OUTER JOIN`, чтобы включить издательства, без совпадающих книг, или `FULL OUTER JOIN`, чтобы вернуть оба типа. Синтаксис перехода необходим по той причине, что не все базы данных используют стандартное обозначение для этих соединений.

Наконец, символы `_` и `%` имеют специальное значение в операции `LIKE`, чтобы обозначить совпадение с одним символом или последовательностью символов. Стандартного способа их литерального использования не существует. Если вы хотите сопоставить все строки, содержащие символ _, применяйте следующую конструкцию:

```
... WHERE ? LIKE %!_% {escape '!'}
```

Здесь мы определили `!` как символ перехода. Комбинация `!_` обозначает литеральный символ подчеркивания.

Множественные результаты

Запрос может вернуть множественные результаты. Это может произойти при выполнении хранимой процедуры или в базах данных, которые позволяют также выполнение множества операторов `SELECT` в одном запросе. Ниже показано, как можно получить все множественные наборы.

1. Используйте метод `execute` для выполнения оператора SQL.
2. Получите первый результат или обновите подсчет.
3. Повторите вызов метода `getMoreResults`, чтобы перейти к следующему результирующему набору. (Этот вызов автоматически закрывает предыдущий результирующий набор.)
4. Процедура завершается, когда заканчиваются результирующие наборы или обновленные подсчеты.

Методы `execute` и `getMoreResults` возвращает значение `true`, если следующим элементом в соединении является результирующий набор. Метод `getUpdateCount` возвращает значение `-1`, если следующим элементом в соединении не является обновление подсчета.

Приведенный ниже цикл проходит по всем результатам:

```
boolean done = false;
boolean isResult = stmt.execute(command);
while (!done)
{
  if (isResult)
  {
    ResultSet result = stmt.getResultSet();
    какие-либо действия с result
  }
```

```
   else
   {
      int updateCount = stmt.getUpdateCount();
      if (updateCount >= 0)
         какие-либо действия с updateCount
      else
         done = true;
   }
   isResult = stmt.getMoreResults();
}
```

API java.sql.ResultSet 1.1

- `boolean getMoreResults()`

  Получает следующий результат для данного оператора. Возвращает значение `true`, если следующий результат существует, и он является результирующим набором.

Получение автоматически генерируемых ключей

Большинство баз данных поддерживают механизм автоматической нумерации строк в базе данных. К сожалению, у разных поставщиков эти механизмы существенно отличаются. Эти автоматические номера часто используются в качестве первичных ключей. Несмотря на то что JDBC не предлагает решения, независимого от поставщиков, для генерирования этих ключей, она предлагает эффективный способ их получения. Когда вы вставляете новую строку в таблицу, и когда происходит автоматическая генерация ключа, вы можете получить их с помощью следующего кода:

```
stmt.executeUpdate(insertStatement, Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();
if (rs.next())
{
   int key = rs.getInt(1);
   . . .
}
```

API java.sql.ResultSet 1.1

- `boolean execute(String statement, int autogenerated)` 1.4
- `int executeUpdate(String statement, int autogenerated)` 1.4

  Выполняет данный оператор SQL, как было описано ранее. Если `autogenerated` имеет значение `Statement.RETURN_GENERATED_KEYS`, а в качестве оператора используется `INSERT`, то первый столбец содержит автоматически сгенерированный ключ.

Прокручиваемые и обновляемые результирующие наборы

Как вы уже знаете, метод `next()` класса `ResultSet` позволяет последовательно перебирать строки, полученные в результате выполнения запроса. Его очень удобно использовать для анализа данных. Однако часто требуется предоставить пользователю возможность визуального просмотра результатов выполнения запроса с переходом к предыдущей и следующей строке, например, так, как показано на рис. 4.5. Предусмотренные функции *прокрутки* результатов выполнения запроса позволяют свободно перемещаться не только к предыдущим и следующим записям, но и в произвольную позицию в результирующем наборе.

Более того, при визуальном просмотре результатов выполнения запроса у пользователей часто возникает желание исправить какие-то данные. В JDBC 1 для этого необходимо было применять команду `UPDATE`, а в JDBC 2 достаточно просто отредактировать данные в результирующем наборе — и СУБД автоматически обновит модифицированную информацию.

Прокрутка результирующего набора

По умолчанию результирующие наборы не являются прокручиваемыми или обновляемыми. Для организации прокрутки результатов выполнения запроса необходимо получить объект `Statement` с помощью приведенного ниже способа:

```
Statement stat = conn.createStatement(type, concurrency);
```

Для предварительно подготовленного запроса нужно использовать 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, необходимо знать ответы на ряд вопросов.

- Нужно ли организовать прокрутку результирующего набора? Если это не требуется, используйте значение `ResultSet.TYPE_FORWARD_ONLY`.
- Если все же нужно организовать прокрутку результатов выполнения запроса, то в таком случае выясните, должен ли результирующий набор отражать те данные, которые были изменены в базе данных уже после выполнения этого запроса? (Далее предполагается, что установлен параметр `ResultSet.TYPE_SCROLL_INSENSITIVE`, т.е. результирующий набор не "реагирует" на изменения в базе данных, которые произошли после выполнения этого запроса.)
- Нужно ли организовать редактирование результатов выполнения запроса и обновление базы данных? (Более подробно эта тема рассматривается в следующих разделах.)

Для организации прокрутки результатов выполнения запроса без возможности редактирования данных можно использовать следующую команду:

```
Statement stat = conn.createStatement(
   ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
```

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 (по умолчанию)
`CONCUR_UPDATABLE`	С редактированием и обновлением базы данных

Теперь можно прокручивать все результаты выполнения запроса, возвращаемые следующим методом:

```
ResultSet rs = stat.executeStat(query);
```

Такой результирующий набор запроса имеет *курсор* для определения текущей позиции.



Не все драйверы баз данных поддерживают работу с курсором прокрутки или обновления. (Методы `supportsResultSetType()` и `supportsResultSetConcurrency()` класса `DatabaseMetaData` сообщают о типах и режимах, которые поддерживаются в данной СУБД.) Но даже если СУБД поддерживает работу во всех описанных режимах, в некоторых запросах нельзя получить результат в соответствии со всеми заданными свойствами. (Например, результат выполнения сложного запроса может оказаться необновляемым.) В этом случае метод `executeQuery()` возвращает объект `ResultSet` с меньшими возможностями и с *предупреждением* `SQLWarning`, подключенным к объекту соединения. (Способ получения предупреждений показан в разделе "Анализ SQL-исключений" ранее в главе.) Кроме того, для поиска фактически используемого режима работы можно применять методы `getType()` и `getConcurrency()` класса `ResultSet`. Иногда отсутствие проверки фактического режима работы приводит к использованию неподдерживаемой операции, например, к применению метода `previous()` для непрокручиваемого результата выполнения запроса. В таком случае неизбежно возникнет исключение `SQLException`.

Прокрутка организована очень просто. Например, для перехода к предыдущим записям используется следующая конструкция:

```
if (rs.previous()) ...
```

Этот метод возвращает значение `true`, если курсор располагается в какой-то конкретной строке, и значение `false`, если курсор располагается перед первой строкой.

Для перемещения курсора на *n* строк вперед или назад используется метод `rs.relative(n)`. Для положительных значений *n* курсор перемещается вперед, а для отрицательных — назад (значение, равное нулю, не приводит к каким-либо перемещениям). При попытке перемещения курсора за пределы имеющегося результата выполнения запроса он располагается либо вслед за последней, либо перед первой записью, в зависимости от знака *n*. После этого курсор возвращает значение `false` и останавливается. Этот метод возвращает значение `true` только при расположении курсора на фактической строке.

Кроме того, курсор можно расположить на строке с номером *n* с помощью метода `rs.absolute(n)`, а получить текущий номер строки *n* можно с помощью такого метода:

```
int n = rs.getRow();
```

Первая запись в результирующем наборе имеет номер 1. Если возвращаемое значение равно 0, значит, курсор находится не в какой-то строке, а либо после последней, либо перед первой записью.

Для перемещения курсора к первой или последней записи, размещения его перед первой или вслед за последней записью предусмотрены методы `first()`, `last()`, `beforeFirst()` и `afterLast()`. А для проверки расположения курсора у первой или последней записи, перед первой или вслед за последней записью предусмотрены методы `isFirst()`, `isLast()`, `isBeforeFirst()` и `isAfterLast()`.

Как видите, прокрутка результатов выполнения запроса организована довольно просто. Все операции, связанные с кэшированием данных, выполняются драйвером базы данных.

Обновляемые результирующие наборы

Для организации редактирования данных, полученных при выполнении запроса, и автоматического обновления базы данных необходимо создать обновляемый результирующий набор. Вовсе не обязательно, чтобы такой набор допускал прокрутку, но обычно обновляемые результирующие наборы создаются с поддержкой прокрутки.

Для получения обновляемого результирующего набора нужно использовать следующую конструкцию:

```
Statement stat = conn.createStatement(
  ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
```

После этого метод `executeQuery()` возвратит обновляемый результирующий набор.



Не все запросы возвращают обновляемый набор. Например, запрос с соединением нескольких таблиц не всегда может быть обновляемым. Это всегда возможно только для запросов на основе одной таблицы или запросов, выполняющих соединение нескольких таблиц по первичным ключам. Для проверки текущего режима работы рекомендуется использовать метод `getConcurrency()` класса `ResultSet`.

Предположим, например, что вы хотите увеличить цену некоторых книг, но у вас нет единого критерия, который можно было бы использовать в команде `UPDATE`. Поэтому вам приходится перебирать в цикле все книги и изменять цены, исходя из текущих условий:

```
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(); // updateRow должен вызываться после обновления полей
  }
}
```

Для всех типов данных SQL предусмотрены соответствующие методы `updateXxx()`, например, `updateDouble()`, `updateString()` и т.д. Как и при работе с методами `getXxx()`, в качестве параметров этих методов могут использоваться номер или имя столбца, а после него указывается новое значение.



При использовании методов updateXxx() следует иметь в виду, что первый параметр с номером столбца означает номер в *результирующем наборе*, который может отличаться от номера столбца в базе данных.

Методы updateXxx изменяют только значения в текущей строке результирующего набора, а не базы данных. Для обновления всех данных из отредактированной строки в базе нужно вызвать метод updateRow(). При перемещении курсора в следующую строку без вызова метода updateRow() все текущие обновления строки будут отменены, поскольку они не были переданы базе данных. Для отмены обновлений в базе данных можно использовать метод cancelRowUpdates().

В предыдущем примере было продемонстрировано изменение существующей строки. Для создания новой строки в базе данных нужно сначала вызвать метод moveToInsertRow(), переместив тем самым курсор в специальную позицию, а именно в *строку вставки* (insert row). После этого новая строка создается в данной позиции с помощью методов updateXxx(). Затем для вставки новой строки в базу данных используется метод insertRow(). По окончании вставки применяется метод moveToCurrentRow() для перемещения курсора назад в позицию, которую он занимал до вызова метода moveToInsertRow(). Ниже показано, как выглядит весь процесс в целом.

```
rs.moveToInsertRow();
rs.updateString("Title", title);
rs.updateString("ISBN", isbn);
rs.updateString("Publisher_Id", pubid);
rs.updateDouble("Price", price);
rs.insertRow();
rs.moveToCurrentRow();
```

Учтите, что никоим образом нельзя задать *расположение* вновь созданной строки в результате выполнения запроса или в базе данных.

Если вы не определите значение столбца в строке вставки, то там будет стоять NULL. Тем не менее, если столбец имеет ограничение NOT NULL, будет сгенерировано исключение, и строка не будет вставлена.

Наконец, для удаления строки используется метод

```
rs.deleteRow();
```

Этот метод немедленно удаляет строку из результирующего набора и из базы данных.

Таким образом, методы updateRow(), insertRow() и deleteRow() класса ResultSet предоставляют те же возможности, что и SQL-команды UPDATE, INSERT и DELETE. Для программистов, работающих на языке Java, вызов методов более привычен, и они предпочитают данный подход использованию SQL-команд.



При неаккуратном использовании обновляемых результирующих наборов можно получить чрезвычайно неэффективный код. Часто применение команды UPDATE позволяет достигнуть *гораздо* большей эффективности, чем создание запроса и просмотр результирующего набора. Обработку обновляемых результирующих наборов имеет смысл использовать в интерактивных программах, в которых пользователь может вносить произвольные изменения. Если же изменениями управляет логика программы, удобнее применять SQL-команду UPDATE.

API java.sql.Connection 1.1

- `Statement createStatement(int type, int concurrency)` 1.2
- `PreparedStatement prepareStatement(String command, int type, int concurrency)`

  Создает SQL-команду или предварительно подготовленную команду и возвращает результирующий набор. Параметры определяют, должен ли результирующий набор поддерживать прокрутку и редактирование.

  Параметры: `command` Предварительно подготовленная команда.

  `type` Одна из констант (`TYPE_FORWARD_ONLY`, `CONCUR_UPDATABLE` или `TYPE_SCROLL_SENSITIVE`), определенная в интерфейсе `ResultSet`.

  `concurrency` Одна из констант (`CONCUR_READ_ONLY` или `CONCUR_UPDATABLE`), определенная в интерфейсе `ResultSet`.

API java.sql.ResultSet 1.1

- `int getType()` 1.2

  Возвращает константу (`TYPE_FORWARD_ONLY`, `CONCUR_UPDATABLE` или `TYPE_SCROLL_SENSITIVE`), обозначающую тип результирующего набора.

- `int getConcurrency()` 1.2

  Возвращает константу (`CONCUR_READ_ONLY` или `CONCUR_UPDATABLE`), обозначающую способ редактирования и обновления результирующего набора.

- `boolean previous()` 1.2

  Перемещает курсор к предыдущей строке. Он возвращает значение `true`, если в результате курсор располагается на записи, и `false`, если он помещается перед первой строкой.

- `int getRow()` 1.2

  Возвращает номер текущей строки. Нумерация строк начинается с 1.

- `boolean absolute(int r)` 1.2

  Перемещает курсор к строке с номером `r`. Он возвращает значение `true`, если в результате курсор указывает на запись.

- `boolean relative(int d)` 1.2

  Перемещает курсор на `d` строк. Если `d` меньше нуля, то перемещение происходит в обратном направлении. Метод возвращает значение `true`, если в результате его выполнения курсор располагается на записи.

- `boolean first()` 1.2
- `boolean last()` 1.2

  Перемещает курсор к первой или к последней записи. Метод возвращает значение `true`, если в результате его выполнения курсор располагается на записи.

- `void beforeFirst()` 1.2
- `void afterLast()` 1.2

  Размещает курсор перед первой или вслед за последней записью.

- `boolean isFirst()` 1.2
- `boolean isLast()` 1.2

  Проверяет, указывает ли курсор на первую или на последнюю запись.

- `boolean isBeforeFirst()` 1.2
- `boolean isAfterLast()` 1.2

  Проверяет, находится ли курсор перед первой записью или после последней.

- `void moveToInsertRow()` 1.2

  Перемещает курсор в *строку вставки*. Строкой вставки называется специальная строка, которая применяется для вставки новых данных с помощью методов `updateXxx()` и `innsertRow()`.

- `void moveToCurrentRow()` 1.2

  Перемещает курсор из строки вставки в то положение, в котором он находился до вызова метода `moveToInsertRow()`.

- `void insertRow()` 1.2

  Включает содержимое строки вставки в базу данных и в результирующий набор.

- `void deleteRow()` 1.2

  Удаляет текущую строку из базы данных и из 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.)

  Обновляет содержимое указанного столбца текущей строки в результирующем наборе.

- `void updateRow()` 1.2

  Передает изменения текущей строки в базу данных.

- `void cancelRowUpdates()` 1.2

  Отменяет изменения текущей строки.

API `java.sql.DatabaseMetaData 1.1`

- `boolean supportsResultSetType(int type)` 1.2

  Возвращает значение `true`, если база данных может поддерживать заданный тип результирующего набора.

  Параметр: `type` Одна из констант (`TYPE_FORWARD_ONLY`, `CONCUR_UPDATABLE` или `TYPE_SCROLL_SENSITIVE`), определенная в интерфейсе `ResultSet` и обозначающая способ прокрутки.

- `boolean supportsResultSetConcurrency(int type, int concurrency)` 1.2

  Возвращает значение `true`, если база данных может поддерживать заданный тип прокрутки и обновления результирующего набора.

  Параметры: `type` Одна из констант (`TYPE_FORWARD_ONLY`, `CONCUR_UPDATABLE` или `TYPE_SCROLL_SENSITIVE`), определенная в интерфейсе `ResultSet` и обозначающая способ прокрутки.

  `concurrency` Одна из констант (`CONCUR_READ_ONLY` или `CONCUR_UPDATABLE`), определенная в интерфейсе `ResultSet` и обозначающая способ обновления.

Наборы строк

Прокручиваемые результирующие наборы предлагают богатые возможности, однако они не свободны от недостатков. В течение всего времени взаимодействия с пользователем соединение с базой данных должно быть открыто. Однако пользователь может отлучиться на длительное время, а открытое соединение будет потреблять ресурсы. В подобной ситуации целесообразно использовать *набор строк* (row set). Интерфейс `RowSet` расширяет интерфейс `ResultSet`, но набор строк не привязан к соединению с базой данных.

Наборы строк также применимы в случае, если требуется переместить результаты выполнения запроса на другой уровень сложного приложения или на другое устройство, например, на мобильный телефон. Переместить результирующий набор нельзя, так как он связан с соединением, кроме того, размеры структуры данных могут быть очень велики.

Ниже перечислены интерфейсы, содержащиеся в пакете `javax.sql.rowset` и расширяющие интерфейс `RowSet`.

- `CachedRowSet` позволяет выполнять некоторые операции при отсутствии соединения. Кэшируемые наборы строк рассматриваются в следующем разделе.
- `WebRowSet` представляет собой кэшируемый набор строк, который может быть сохранен в XML-файле. XML-файл может быть передан другому компоненту приложения и открыт с помощью другого объекта `WebRowSet`.
- Интерфейсы `FilteredRowSet` и `JoinRowSet` поддерживают "легковесные" операции с наборами строк, эквивалентные SQL-командам `SELECT` и `JOIN`. Эти операции базируются только на данных, содержащихся в наборе строк, и соединение с базой не устанавливается.
- `JdbcRowSet` представляет собой оболочку для `ResultSet`. Он определяет методы доступа и модифицирующие методы, превращая результирующий набор в компонент JavaBeans. (Более подробно компоненты JavaBeans рассматриваются в главе 8.)

В компании Sun Microsystems предполагают, что производители баз данных предоставят эффективные реализации этих интерфейсов. К тому же существуют базовые реализации, которые позволяют использовать наборы строк даже в том случае, если в конкретной базе данных они не поддерживаются. Эти базовые реализации принадлежат пакету `com.sun.rowset`. Имена классов заканчиваются символами `Impl`, например, `CachedRowSetImpl`.

Кэшируемые наборы строк

Кэшируемый набор строк содержит данные из результирующего набора. Поскольку интерфейс `CachedRowSet` расширяет `ResultSet`, вы можете использовать кэшируемый набор строк точно так же, как и результирующий набор. Однако наборы строк имеют существенное преимущество: вы можете закрыть соединение с базой и продолжать работать с набором. Как вы увидите из программы, представленной в листинге 4.4, такая возможность существенно упрощает создание интерактивных приложений. При получении команды от пользователя открывается соединение, выполняется запрос, результаты помещаются в набор строк, после чего соединение с базой данных закрывается.

Кэшируемый набор строк даже позволяет модифицировать содержащиеся в нем данные. Конечно, результаты изменений не отражаются в базе немедленно. Для того чтобы принять накопленные изменения, необходимо выполнить специальный запрос. В результате объект CachedRowSet повторно устанавливает соединение и выполняет SQL-команды для записи изменений.

Объект CachedRowSet заполняется данными из результирующего набора.

```
ResultSet result = ...;
CachedRowSet crs = new com.sun.rowset.CachedRowSetImpl();
   // Можно также использовать реализацию, предложенную поставщиком базы данных.
rowset.populate(result);
conn.close(); // Теперь можно закрыть соединение с базой.
```

Можно также предоставить объекту CachedRowSet автоматически установить соединение. Установите следующие параметры базы данных:

```
rowset.setURL("jdbc:derby://localhost:1527/COREJAVA");
rowset.setUsername("dbuser");
rowset.setPassword("secret");
```

Затем создайте команду:

```
crs.setCommand("SELECT * FROM Books WHERE PUBLISHER = ?");
crs.setString(1, publisherName);
```

И, наконец, заполните набор строк результатами запроса:

```
crs.execute();
```

В результате этого вызова будет установлено соединение с базой данных, выполнен запрос, заполнен набор строк и разорвано соединение.

Если полученный результат запроса имеет слишком большой объем, вы можете не помещать его полностью в набор строк. В конце концов, ваши пользователи, скорее всего, взглянут всего лишь на некоторые строки. В этом случае необходимо определить размер страницы:

```
CachedRowSet crs = ...;
crs.setCommand(command);
crs.setPageSize(20);
...
crs.execute();
```

Теперь у вас будет только 20 строк. Чтобы получить следующую порцию строк, вызовите метод

```
crs.nextPage();
```

Для просмотра и модификации набора строк используются те же методы, что и при работе с результирующим набором. Если вы измените содержимое набора, то для записи изменений в базу данных необходимо использовать один из следующих вызовов:

```
crs.acceptChanges(conn);
crs.acceptChanges();
```

Второй вариант обращения к методу действует только в том случае, если вы зададите для набора строк все данные, необходимые для соединения с базой данных (URL, имя пользователя и пароль).

Вы уже знаете из раздела “Обновляемые результирующие наборы”, что 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 при записи в базу проблем не возникает.



Если вы заполните набор строк данными из результирующего набора, набору строк не будет известно имя таблицы, подлежащей обновлению. В этом случае нужно специально указать имя таблицы, вызвав метод `setTable()`.

Если данные в базе изменились с того момента, как набор строк был заполнен информацией, то возникают дополнительные сложности, связанные с несоответствием данных. В базовой реализации проверяется, совпадают ли исходные значения набора строк (т.е. значения перед редактированием) с текущими значениями базы. Если проверка дает положительный результат, содержимое базы заменяется модифицированными данными. В противном случае генерируется исключение `SyncProviderException` и изменения не записываются. В других реализациях могут использоваться другие способы синхронизации данных.



javax.sql.RowSet 1.4

- `String getURL()`
- `void setURL(String url)`

  Возвращает или устанавливает URL базы данных.

- `String getUsername()`
- `void setUsername(String username)`

  Возвращает или устанавливает имя пользователя для соединения с базой данных.

- `String getPassword()`
- `void setPassword(String password)`

  Возвращает или устанавливает пароль для соединения с базой данных.

- `String getCommand()`
- `void setCommand(String command)`

  Возвращает или устанавливает команду, при выполнении которой осуществляется заполнение набора строк данными.

- `void execute()`

  Заполняет строку, выполняя команду, установленную с помощью метода `setCommand()`. Для того чтобы диспетчер драйверов мог установить соединение, должны быть заданы URL, пользовательское имя и пароль.



javax.sql.rowset.CachedRowSet 5.0

- `void execute(Connection conn)`

  Заполняет набор строк, выполняя команду, установленную с помощью метода `setCommand()`. Данный метод использует указанное соединение и закрывает его.

- `void populate(ResultSet result)`

  Заполняет кэшируемый набор строк данными из указанного результирующего набора.

- `String getTableName()`
- `void setTableName(String tableName)`

  Возвращает или устанавливает имя таблицы, данными из которой заполняется кэшируемый набор строк.

- `int getPageSize()`
- `void setPageSize(int size)`

  Возвращает или устанавливает размер страницы.

- `boolean nextPage()`
- `boolean previousPage()`

  Загружает следующую или предыдущую страницу строк. Возвращает значение `true`, если существует следующая или предыдущая страница.

- `void acceptChanges()`
- `void acceptChanges(Connection conn)`

  Повторно устанавливает соединение с базой данных и записывает изменения, внесенные в набор строк. Если с момента заполнения набора содержимое базы было изменено, данные не могут быть записаны. В этом случае генерируется исключение `SyncProviderException`.

Метаданные

В предыдущих разделах рассматривались способы ввода и обновления данных в таблицах базы и создания запросов. Кроме того, в JDBC предусмотрены дополнительные возможности для получения информации о *структуре* таблиц и самой базы данных. Например, можно получить список всех таблиц базы данных либо имена всех столбцов с типами данных в них. Эти сведения вряд ли будут очень полезны при создании приложения для работы с какой-то конкретной базой данных, потому что в таких случаях ее структура точно известна. Однако они пригодятся тем разработчикам, которые создают свои программные продукты для работы с произвольными базами данных.

В языке SQL информация о структуре базы и ее компонентов называется *метаданными* (metadata). Такое название выбрано лишь для того, чтобы отличать данные о базе от основных данных. Существуют метаданные трех типов, описывающие структуру базы данных, структуру результирующих наборов и параметры предварительно подготовленных команд.

Для поиска более подробных сведений нужен объект `DatabaseMetaData`, который можно получить на основе соединения с базой данных:

```
DatabaseMetaData meta = conn.getMetaData();
```

Теперь можно приступать непосредственно к получению метаданных. Например, вызов представленного ниже метода приводит к получению результирующего набора, содержащего информацию обо всех таблицах базы данных. (Параметры этого метода рассматриваются ниже при описании API.)

```
ResultSet mrs = meta.getTables(null, null, null, new String[] { "TABLE" });
```

Каждая строка этого результирующего набора содержит сведения об отдельной таблице. В данном случае нас интересует только третий столбец, т.е. имя таблицы. Его дает вызов `rs.getString(3)`. Ниже представлен фрагмент кода, предназначенный для заполнения раскрывающегося списка:

```
while (mrs.next())
   tableNames.addItem(mrs.getString(3));
```

Базы данных могут иметь очень сложную структуру, а стандарт SQL предоставляет довольно много возможностей. В классе `DatabaseMetaData` предусмотрено всего более сотни разных методов, которые можно использовать для получения информации о структуре базы данных. Ниже показаны примеры таких методов:

```
meta.supportsCatalogsInPrivilegeDefinitions()
meta.nullPlusNonNullIsNull()
```

Судя по названиям этих методов, они предназначены только для очень опытных программистов, в частности, для создания переносимого кода, который способен работать с разными типами баз данных.

Информация о структуре базы данных хранится в классе `DatabaseMetaData`, а сведения о структуре результирующих наборов — в классе `ResultSetMetaData`. После получения результатов выполнения запроса можно определить количество столбцов, имена столбцов, типы данных в них и ширину полей. Ниже показан типичный цикл:

```
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 рассматриваются способы создания простого инструмента, предназначенного для просмотра и анализа структуры базы данных — код этой программы представлен в листинге 4.4. Эта программа показывает также использование кэшированного набора строк.

Окно создаваемой программы содержит раскрывающийся список с именами всех таблиц базы данных. Как показано на рис. 4.8, после выбора какой-то одной таблицы в центральной части фрейма будут представлены имена столбцов из этой таблицы, а также значения из первой записи. Для просмотра записей нужно щелкать на кнопке Next (Следующая) и Previous (Предыдущая). Кроме этого, вы можете также удалять строки и редактировать значения в них. Чтобы сохранить изменения в базе данных, щелкните на кнопке Save (Сохранить).

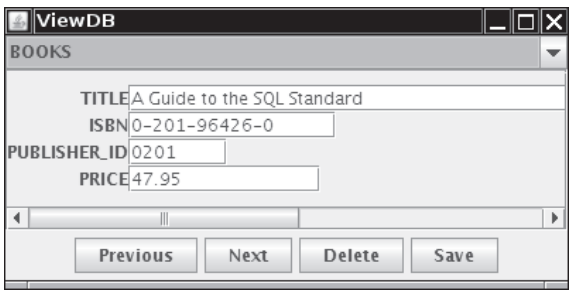


Рис. 4.8. Приложение `ViewDB`



В состав баз данных обычно включаются инструменты для просмотра и редактирования таблиц, предоставляющие гораздо более богатые возможности. Если для вашей базы такого инструмента нет, попробуйте использовать iSQL-Viewer (`http://isql.sourceforge.net`) или SQuirreL (`http://squirrel-sql.sourceforge.net`). Эти программы позволяют просматривать таблицы любой JDBC-базы. Наша программа отнюдь не претендует на то, чтобы конкурировать с этими инструментами; она лишь демонстрирует общие принципы создания программ для работы с произвольными таблицами.

Листинг 4.4. Содержимое файла `ViewDB.java`

```
1.  import com.sunrowset.*;
2.  import java.sql.*;
3.  import java.awt.*;
4.  import java.awt.event.*;
5.  import java.io.*;
6.  import java.util.*;
7.  import javax.swing.*;
8.  import javax.sql.*
9.  import javax.rowset;
10.
11. /**
12.  * Эта программа демонстрирует использование
13.  * метаданных для отображения таблиц базы данных.
14.  * @version 1.31 2007-06-28
15.  * @author Cay Horstmann
16.  */
17. public class ViewDB
18. {
19.    public static void main(String[] args)
20.    {
21.       EventQueue.invokeLater(new Runnable()
22.       {
23.          public void run()
24.          {
25.             JFrame frame = new ViewDBFrame();
26.             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27.             frame.setVisible(true);
28.          }
29.       });
30.    }
31. }
32.
33. /**
34.  * Фрейм, содержащий панель с данными
35.  * и кнопки перемещения по записям.
36.  */
37. class ViewDBFrame extends JFrame
38. {
39.    public ViewDBFrame()
40.    {
41.       setTitle("ViewDB");
42.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
43.
44.       tableNames = new JComboBox();
45.       tableNames.addActionListener(new ActionListener()
46.          {
47.             public void actionPerformed(ActionEvent event)
48.             {
49.                showTable((String) tableNames.getSelectedItem());
50.             }
51.          });
52.       add(tableNames, BorderLayout.NORTH);
53.
54.       try
55.       {
```

```
56.         readDatabaseProperties();
57.         Connection conn = getConnection();
58.         try
59.         {
60.            DatabaseMetaData meta = conn.getMetaData();
61.            ResultSet mrs = meta.getTables(null, null, null, new String[] { "TABLE" });
62.            while (mrs.next())
63.               tableNames.addItem(mrs.getString(3));
64.         }
65.         finally
66.         {
67.            conn.close();
68.         }
69.      }
70.      catch (SQLException e)
71.      {
72.         JOptionPane.showMessageDialog(this, e);
73.      }
74.      catch (IOException e)
75.      {
76.         JOptionPane.showMessageDialog(this, e);
77.      }
78.
79.      JPanel buttonPanel = new JPanel();
80.      add(buttonPanel, BorderLayout.SOUTH);
81.
82.         previousButton = new JButton("Previous");
83.         previousButton.addActionListener(new ActionListener()
84.            {
85.               public void actionPerformed(ActionEvent event)
86.               {
87.                  showPreviousRow();
88.               }
89.            });
90.         buttonPanel.add(previousButton);
91.      }
92.
93.      nextButton = new JButton("Next");
94.      nextButton.addActionListener(new ActionListener()
95.         {
96.            public void actionPerformed(ActionEvent event)
97.            {
98.               showNextRow();
99.            }
100.        });
101.     buttonPanel.add(nextButton);
102.
103.     deleteButton = new JButton("Delete");
104.     deleteButton.addActionListener(new ActionListener()
105.        {
106.        public void actionPerformed(ActionEvent event)
107.           {
108.              deleteRow();
109.           }
110.        });
111.     buttonPanel.add(deleteButton);
112.
```

```
113.      saveButton = new JButton("Save");
114.      saveButton.addActionListener(new ActionListener()
115.        {
116.          public void actionPerformed(ActionEvent event)
117.          {
118.             saveChanges();
119.          }
120.        });
121.      buttonPanel.add(saveButton);
122.    }
123.
124.    /**
125.     * Подготовка текстовых полей к отображению новой таблицы и показ первой строки.
126.     * @param tableName the name of the table to display
127.     */
128.    public void showTable(String tableName)
129.    {
130.      try
131.      {
132.        // открытие соединения
133.        Connection conn = getConnection();
134.        try
135.        {
136.          // получение результирующего набора
137.          Statement stat = conn.createStatement();
138.          ResultSet result = stat.executeQuery("SELECT * FROM " + tableName);
139.          // копирование в кэшированный результирующий набор
140.          crs = new CachedRowSetImpl();
141.          crs.setTableName(tableName);
142.          crs.populate(result);
143.        }
144.        finally
145.        {
146.          conn.close();
147.        }
148.
149.        if (scrollPane != null) remove(scrollPane);
150.        dataPanel = new DataPanel(crs);
151.        scrollPane = new JScrollPane(dataPanel);
152.        add(scrollPane, BorderLayout.CENTER);
153.        validate();
154.        showNextRow();
155.      }
156.      catch (SQLException e)
157.      {
158.        JOptionPane.showMessageDialog(this, e);
159.      }
160.    }
161.
162.    /**
163.     * Переход к предыдущей строке таблицы.
164.     */
165.    public void showPreviousRow()
166.    {
167.      try
168.      {
169.        if (crs == null || crs.isFirst()) return;
```

```
170.        crs.previous();
171.        dataPanel.showRow(crs);
172.     }
173.     catch (SQLException e)
174.     {
175.        for (Throwable t : e)
176.           t.printStackTrace();
177.     }
178.  }
179.
180.  /**
181.   * Переход к следующей строке таблицы.
182.   */
183.  public void showNextRow()
184.  {
185.     try
186.     {
187.        if (crs == null || crs.isLast()) return;
188.        crs.next();
189.        dataPanel.showRow(crs);
190.     }
191.     catch (SQLException e)
192.     {
193.        JOptionPane.showMessageDialog(this, e);
194.     }
195.  }
196.
197.  /**
198.   * Удаление текущей строки таблицы.
199.   */
200.  public void deleteRow()
201.  {
202.     try
203.     {
204.        Connection conn = getConnection();
205.        try
206.        {
207.           crs.deleteRow();
208.           crs.acceptChanges(conn);
209.           if (!crs.isLast()) crs.next();
210.           else if (!crs.isFirst()) crs.previous();
211.           else crs = null;
212.           dataPanel.showRow(crs);
213.        }
214.        finally
215.        {
216.           conn.close();
217.        }
218.     }
219.     catch (SQLException e)
220.     {
221.        JOptionPane.showMessageDialog(this, e);
222.     }
223.  }
224.
```

```
225.    /**
226.     * Сохранение всех изменений.
227.     */
228.    public void saveChanges()
229.    {
230.       try
231.       {
232.          Connection conn = getConnection();
233.          try
234.          {
235.             dataPanel.setRow(crs);
236.             crs.acceptChanges(conn);
237.          }
238.          finally
239.          {
240.             conn.close();
241.          }
242.       }
243.       catch (SQLException e)
244.       {
245.          JOptionPane.showMessageDialog(this, e);
246.       }
247.    }
248.    private void readDatabaseProperties() throws IOException
249.    {
250.       props = new Properties();
251.       FileInputStream in = new FileInputStream("database.properties");
252.       props.load(in);
253.       in.close();
254.       String drivers = props.getProperty("jdbc.drivers");
255.       if (drivers != null) System.setProperty("jdbc.drivers", drivers);
256.    }
257.    /**
258.     * Получение соединения из свойств, определенных в файле database.properties
259.     * @return соединение с базой данных
260.     */
261.    private Connection getConnection() throws SQLException
262.    {
263.       String url = props.getProperty("jdbc.url");
264.       String username = props.getProperty("jdbc.username");
265.       String password = props.getProperty("jdbc.password");
266.
267.       return DriverManager.getConnection(url, username, password);
268.    }
269.    public static final int DEFAULT_WIDTH = 400;
270.    public static final int DEFAULT_HEIGHT = 200;
271.
272.    private JButton previousButton;
273.    private JButton nextButton;
274.    private JButton deleteButton;
275.    private JButton saveButton;
276.    private DataPanel dataPanel;
277.    private Component scrollPane;
278.    private JComboBox tableNames;
279.    private Properties props;
280.    private CachedRowSet crs;
281. }
```

```
282. /**
283.  * Эта панель отображает содержимое результирующего набора.
284.  */
285. class DataPanel extends JPanel
286. {
287.    /**
288.     * Создание панели данных.
289.     * @param rs результирующий набор, содержимое которого будет
290.     * отображаться на этой панели
291.     */
292.    public DataPanel(RowSet rs) throws SQLException
293.    {
294.       fields = new ArrayList<JTextField>();
295.       setLayout(new GridBagLayout());
296.       GridBagConstraints gbc = new GridBagConstraints();
297.       gbc.gridwidth = 1;
298.       gbc.gridheight = 1;
299.
300.       ResultSetMetaData rsmd = rs.getMetaData();
301.       for (int i = 1; i <= rsmd.getColumnCount(); i++)
302.       {
303.          gbc.gridy = i - 1;
304.
305.          String columnName = rsmd.getColumnLabel(i);
306.          gbc.gridx = 0;
307.          gbc.anchor = GridBagConstraints.EAST;
308.          add(new JLabel(columnName), gbc);
309.
310.          int columnWidth = rsmd.getColumnDisplaySize(i);
311.          JTextField tb = new JTextField(columnWidth);
312.          if (!rsmd.getColumnClassName(i).equals("java.lang.String"))
313.             tb.setEditable(false);
314.
315.          fields.add(tb);
316.
317.          gbc.gridx = 1;
318.          gbc.anchor = GridBagConstraints.WEST;
319.          add(tb, gbc);
320.       }
321.    }
322.
323.    /**
324.     * Отображение строки базы данных посредством заполнения
325.     * всех текстовых полей значениями столбцов.
326.     */
327.    public void showRow(ResultSet rs) throws SQLException
328.    {
329.       for (int i = 1; i <= fields.size(); i++)
330.       {
331.          String field = rs.getString(i);
332.          JTextField tb = (JTextField) fields.get(i - 1);
333.          tb.setText(field);
334.       }
335.    }
336.
```

```
337.    /**
338.     * Обновление измененных данных в текущей строке результирующего набора.
339.     */
340.    public void setRow(RowSet rs) throws SQLException
341.    {
342.       for (int i = 1; i <= fields.size(); i++)
343.       {
344.          String field = rs.getString(i);
345.          JTextField tb = (JTextField) fields.get(i - 1);
346.          if (!field.equals(tb.getText()))
347.             rs.updateString(i, tb.getText());
348.       }
349.       rs.updateRow();
350.    }
351.
352.    private ArrayList<JTextField> fields;
353. }
```

API java.sql.Connection 1.1

- `DatabaseMetaData getMetaData()`

  Возвращает метаданные для соединения в виде объекта `DatabaseMetaData`.

API java.sql.DatabaseMetaData 1.1

- `ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String types[])`

  Предоставляет описание всех таблиц в каталоге, которые соответствуют шаблонам схемы и имен таблиц, а также заданным типам. (*Схемой* (schema) называется группа связанных таблиц и полномочий доступа. *Каталогом* (catalog) называется связанная группа схем. Эти понятия важны для структурирования больших баз данных.)

  Параметры `catalog` и `schemaPattern` могут быть пустыми строками (`""`) для извлечения таблиц без каталога и схемы либо иметь значения `null`, если нужно возвращать таблицы без учета каталога или схемы.

  Массив `types` содержит имена типов таблиц: `TABLE`, `VIEW`, `SYSTEM TABLE`, `GLOBAL TEMPORARY`, `LOCAL TEMPORARY`, `ALIAS` и `SYNONYM`. Если для `types` указано значение `null`, будут возвращены таблицы всех типов.

  Результаты выполнения запроса содержат пять столбцов, которые имеют тип `String` и перечислены в табл. 4.8.

Таблица 4.8. Результирующий набор метода `getTables()`

Столбец	Имя	Описание
1	`TABLE_CAT`	Каталог таблиц (значение может быть равно `null`)
2	`TABLE_SCHEM`	Схема (значение может быть равно `null`)
3	`TABLE_NAME`	Имя таблицы
4	`TABLE_TYPE`	Тип таблицы
5	`REMARKS`	Комментарии для таблицы

- `int getJDBCMajorVersion()` 1.4
- `int getJDBCMinorVersion()` 1.4

  Возвращает основной и дополнительный номер версии драйвера JDBC, с помощью которого установлено 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 могут быть установлены одновременно.

- `int getMaxStatements()`

  Возвращает максимальное количество одновременно открытых команд для одного соединения. Если это количество не ограничено или не известно, возвращается значение 0.

API java.sql.ResultSet 1.1

- `ResultSetMetaData getMetaData()`

  Возвращает метаданные, связанные со столбцами результирующего набора `ResultSet`.

API java.sql.ResultSetMetaData 1.1

- `int getColumnCount()`

  Возвращает количество столбцов для текущего результирующего набора `ResultSet`.

- `int getColumnDisplaySize(int column)`

  Возвращает максимальную ширину столбца, указанного с помощью целочисленного параметра.

  Параметр: `column` Номер столбца

- `int getColumnLabel(int column)`

  Возвращает предполагаемый заголовок для указанного столбца.

  Параметр: `column` Номер столбца

- `int getColumnName(int column)`

  Возвращает имя столбца, указанного с помощью целочисленного параметра.

  Параметр: `column` Номер столбца

Транзакции

Группа команд может быть оформлена в виде *транзакции* (transaction), которая может быть *зафиксирована* (commit) после успешного выполнения всех команд либо отвергнута (rollback), если при выполнении хотя бы одной из команд произойдет какая-то ошибка.

Основная причина использования транзакций заключается в поддержании *целостности базы данных*. Предположим, например, что вы хотите перевести средства с одного банковского счета на другой. В этом случае нужно одновременно снять деньги с одного счета и пополнить ими другой. Если при обновлении одного из счетов произойдет ошибка, операция с другим должна быть отменена.

Говорят, что транзакция *зафиксирована* (commit), если успешно выполнены все команды транзакции. В противном случае она *отвергается*, или *производится откат* (rollback), т.е. отменяются все изменения в базе данных, которые выполнялись после предыдущей зафиксированной транзакции.

По умолчанию соединение с базой данных обладает возможностью *автоматической фиксации* (autocommit mode), т.е. каждая SQL-команда фиксируется после ее успешного выполнения. Причем зафиксированную команду нельзя подвергнуть откату. Для отключения режима автоматической фиксации используется метод

```
conn.setAutoCommit(false);
```

Отключив автоматическую фиксацию, можно приступать к созданию объекта `Statement`:

```
Statement stat = conn.createStatement();
```

Затем нужное количество раз вызывается метод `executeUpdate()`:

```
stat.executeUpdate(command1);
stat.executeUpdate(command2);
stat.executeUpdate(command3);
...
```

После выполнения всех этих команд необходимо вызвать метод `commit()`:

```
conn.commit();
```

В случае возникновения какой-либо ошибки нужно откатить все предыдущие команды (кроме `commit`) с помощью метода `rollback()`:

```
conn.rollback();
```

Откат обычно производится, если при выполнении транзакции генерируется исключение `SQLException`.

Точки сохранения

Повысить возможности контроля за процессом отката позволяют точки сохранения. При создании точки сохранения помечается позиция, в которую затем можно перейти, не возвращаясь в точку начала транзакции. Рассмотрим в качестве примера приведенный ниже фрагмент кода.

```
Statement stat = conn.createStatement(); // начало транзакции;
   // в эту точку осуществляется переход при вызове rollback()
stat.executeUpdate(command1);
Savepoint svpt = conn.setSavepoint(); // Установка точки сохранения;
   // в нее осуществляется переход при вызове rollback(svpt)
stat.executeUpdate(command2);
if (...) conn.rollback(svpt); // Отмена command2
...
conn.commit();
```

Если точка сохранения не нужна, ее следует отменить:

```
stat.releaseSavepoint(svpt);
```

Пакетные обновления

Предположим, что в программе следует выполнить несколько команд вставки `INSERT` для ввода данных в таблицы. В JDBC 2 повысить производительность программы можно с помощью *пакетного обновления* (batch update). В этом случае обновление выполняется для всей группы из нескольких команд, а не для каждой отдельной команды обновления из этой группы.



Для проверки наличия этой возможности в используемой СУБД предусмотрен метод `supportsBatchUpdates()` класса `DatabaseMetaData`.

Помимо команд управления данными `INSERT`, `UPDATE` и `DELETE`, в группах команд пакетного обновления можно использовать команды определения данных, например `CREATE TABLE` и `DROP TABLE`. Однако в них нельзя использовать команду `SELECT`, так как при этом возникнет исключение. (В концептуальном плане команда `SELECT` не имеет смысла в пакетом режиме, поскольку она возвращает результирующий набор, не обновляя базу данных.)

Для организации пакетного обновления сначала нужно создать объект `Statement`:

```
Statement stat = conn.createStatement();
```

Затем вместо вызова метода `executeUpdate()` потребуется вызвать метод `addBatch()`:

```
String command = "CREATE TABLE ...";
stat.addBatch(command);

while(...)
{
   command = "INSERT INTO ... VALUES (" + ... + ")";
   stat.addBatch(command);
}
```

Наконец, следует выполнить пакетное обновление:

```
int[] counts = stat.executeBatch();
```

Метод `executeBatch()` возвращает массив счетчиков строк, задействованных при выполнении каждой команды.

Для правильной обработки ошибок пакетное обновление необходимо рассматривать как одну транзакцию. При сбое пакетного обновления, очевидно, потребуется выполнить откат к исходному состоянию.

Для организации фиксации и отката такой транзакции следует отключить режим автоматической фиксации, собрать группу команд пакетного обновления, выполнить и зафиксировать их, а затем восстановить режим автоматической фиксации:

```
boolean autoCommit = conn.getAutoCommit();
conn.setAutoCommit(false);
Statement stat = conn.getStatement();
...
// несколько вызовов команды stat.addbatch();
...
stat.executeBatch();
conn.commit();
conn.setAutoCommit(autoCommit);
```



java.sql.Connection 1.1

- `boolean getAutoCommit()`
- `void setAutoCommit(boolean b)`

  Возвращает или устанавливает режим автоматической фиксации; если `b` равно `true`, то все команды будут автоматически фиксироваться сразу после их выполнения.

- `void commit()`
  Фиксирует все команды, которые поступили после последней фиксации.
- `void rollback()`
  Отменяет все изменения, которые были внесены после последней фиксации.
- `Savepoint setSavepoint()` 1.4
- `Savepoint setSavepoint(String name)` 1.4
  Устанавливает неименованную или именованную точку сохранения.
- `void rollback(Savepoint svpt)` 1.4
  Выполняет откат всех команд до указанной точки сохранения.
- `void releaseSavepoint(Savepoint svpt)` 1.4
  Удаляет указанную точку сохранения.

API java.sql.Savepoint 1.4

- `int getSavepointId()`
  Возвращает идентификатор неименованной точки сохранения. Если точка сохранения является именованной, генерируется исключение `SQLException`.
- `String getSavepointName()`
  Возвращает имя точки сохранения. Если точка сохранения является неименованной, генерируется исключение `SQLException`.

API java.sql.Statement 1.1

- `void addBatch(String command)` 1.2
  Включает указанную команду в текущую группу пакетного обновления.
- `int[] executeBatch()` 1.2
  Выполняет все команды пакетного обновления. Каждое значение в полученном массиве соответствует одной из пакетных команд. Если оно является положительным, то это будет счетчик строк. Если это будет значение `SUCCESS_NO_INFO`, значит, команда была выполнена, но счетчик строк не будет доступен. Если это значение `EXECUTE_FAILED`, команда не была выполнена.

API java.sql.DatabaseMetaData 1.1

- `boolean supportsBatchUpdates()` 1.2
  Возвращает значение `true`, если драйвер поддерживает пакетное обновление.

Расширенные типы данных в языке SQL

В табл. 4.9 приведен список типов данных SQL, поддерживаемых JDBC, и их эквиваленты в языке Java.

Тип `ARRAY` языка SQL представляет собой 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` используется метод `getArray()`. В интерфейсе `java.sql.Array` также предусмотрены методы извлечения значений массива.

Таблица 4.9. Типы данных SQL и соответствующие им типы в языке 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), LONGCHAR	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), LOG NVARCHAR	String
NCLOB	java.sql.NClob
SQLXML	java.sql.SQLXML

При получении LOB-объекта или массива из базы данные фактически доставляются только после запроса отдельных значений. Это сделано для повышения эффективности работы. Дело в том, что размер объектов может оказаться очень большим.

Некоторые базы данных поддерживают значения ROWID, которые описывают местонахождение строки, благодаря чему ее можно извлечь очень быстро. JDBC 4 содержит интерфейс java.sql.RowId и предлагает методы для передачи идентификатора (ID) строки в запросы и извлечения его из результатов.

Строка национальных символов (NCHAR — national character string, и его варианты) хранит строки в локальной кодировке символов и сортирует их согласно условиям локальной сортировки. JDBC 4 предлагает методы для преобразования между объектами Java String и строками национальных символов в запросах и результатах.

В некоторых базах данных допускается хранение типов, определенных пользователем. В JDBC 3 поддерживается механизм автоматического отображения структурных типов SQL в Java-объекты.

Некоторые базы данных предлагают естественное хранение данных XML. JDBC 4 предлагает интерфейс SQLXML, который может быть связующим звеном между внутренним представлением XML и интерфейсами DOM Source/Result, а также двоичными потоками. Подробные сведения о классе SQLXML можно найти в документации по API.

Больше в этой главе мы не будем говорить о расширенных типах SQL. Более полную информацию по этому вопросу можно найти в *JDBC API Tutorial and Reference* и в спецификациях JDBC 4.

Управление соединениями в Web- и производственных приложениях

Способ соединения с базой данных с использованием параметров из файла `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 посредством JNDI (Java Naming and Directory Interface). Свойства источников данных организованы в виде каталогов. Это позволяет обеспечить централизованное управление именами пользователей, паролями и URL JDBC.

В такой среде для установления соединения с базой данных рекомендуется использовать приведенный ниже код:

```
Context jndiContext = new InitialContext();
DataSource source = (DataSource) jndiContext.lookup("java:comp/env/jdbc/corejava");
Connection conn = source.getConnection();
```

Обратите внимание, что в этом коде уже не используется диспетчер драйверов `DriverManager`. Вместо него для поиска *источника данных* применяется служба JNDI. Источник данных предоставляет интерфейс, который позволяет устанавливать простые JDBC-соединения, а также выполнять некоторые более сложные функции, например, поддержку распределенных транзакций с несколькими базами данных. Интерфейс `DataSource` содержится в стандартном пакете `javax.sql`.



В контейнере Java EE 5 вам не нужно даже иметь программу в поиске JNDI. Просто используйте аннотацию `Resource` в поле `DataSource`, и во время загрузки вашего приложения будет задан эталонный источник данных:

```
@Resource("jdbc/corejava")
private DataSource source;
```

Очевидно, что источник данных нуждается в настройке. Если вы напишете программу для работы с базой данных, которая будет выполняться под управлением контейнера сервлетов (например, Apache Tomcat) или сервера приложений (например, GlassFish), то информацию о настройке базы (в том числе имя JNDI, JDBC URL, имя пользователя и пароль) целесообразно поместить в конфигурационный файл, или же ее нужно будет задать в графическом интерфейсе администратора.

Управление именами пользователей и регистрационными данными является только одной из проблем, требующих специального внимания. Второй проблемой является стоимость установленных соединений с базами данных. В наших примерах СУБД используются две стратегии для получения соединения с базой данных. Программа `QueryDB`, представленная в листинге 4.3, устанавливает одно соединение с базой данных в самом начале работы и закрывает его по окончании выполнения. Программа `ViewDB`, представленная в листинге 4.4, открывает новое соединение всякий раз, когда в этом возникает необходимость.

Однако ни один из этих подходов не является полностью пригодным. Соединения с базами данных — это конечный ресурс. Если пользователь перестанет работать с приложением в течение некоторого времени, соединение не следует оставлять открытым. И наоборот, получение соединения для каждого запроса с последующим его закрытием обходится очень дорого.

Для решения данной проблемы следует использовать *пул* (pool) соединений. Это значит, что соединения с базами данных не закрываются физически, а хранятся в

очереди и повторно используются для других запросов. Интерфейс JDBC содержит несколько инструментов для организации работы с пулом соединений. Но учтите, что в разных СУБД пул может быть реализован по-разному. Причем он не всегда входит в состав драйвера JDBC. Некоторые поставщики Web-контейнеров и серверов приложений предлагают реализации пулов соединений в составе своих приложений.

Использование пула соединений полностью прозрачно для программиста. На основе информации об источнике данных и с помощью метода `getConnection()` обеспечивается соединение с пулом. По окончании использования этого соединения вызывается метод `close()`. При этом физическое соединение не закрывается, а пулу соединений поступает сообщение о прекращении его использования. Пул соединений обычно старается также подготовить команды.

Теперь мы имеем все основные сведения о работе JDBC, которые необходимы для создания простых приложений, предполагающих взаимодействие с базами данных. Однако, как уже отмечалось в начале главы, базы данных могут иметь очень сложную структуру. Информацию о дополнительных инструментах JDBC можно найти в *JDBC API Tutorial and Reference* или в спецификациях JDBC.

Введение в LDAP

В предшествующих разделах описана организация взаимодействия с реляционной базой данных. Теперь вкратце рассмотрим иерархические базы данных, использующие LDAP (Lightweight Directory Access Protocol — облегченный протокол доступа к каталогам). Этот раздел написан на основе материала, изложенного во втором издании книги Гери и Хорстманна *JavaServer Faces. Библиотека профессионала* (ИД "Вильямс", 2008 г.).

Иерархические базы данных имеют преимущество перед реляционными базами в тех случаях, когда данные, с которыми оперирует приложение, по своей природе имеют древовидную структуру, а также тогда, когда при работе программы операции чтения используются гораздо чаще, чем операции записи. LDAP чаще всего применяется для формирования каталогов, содержащих такие данные, как пользовательские имена, пароли и права доступа.



Для тех, кто желает получить более подробную информацию о LDAP, можем порекомендовать обратиться ко 2-му изданию книги Тимоти Хауэса (Timothy Howes) и др. *Understanding and Deploying LDAP Directory Services* (Addison-Wesley Professional, 2003 г.).

В отличие от реляционных баз, где информация содержится в наборе таблиц, база LDAP организует данные в виде древовидной структуры. Каждая запись в составе древовидной структуры содержит следующие сведения.

- Любое (в том числе нулевое) количество *атрибутов* (attribute). Атрибут состоит из идентификатора и значения. В качестве примера атрибута можно привести выражение `cn=John Q. Public`. (Идентификатор `cn` соответствует "общему имени". Наиболее часто используемые атрибуты LDAP перечислены в табл. 4.10.)
- Один или несколько *объектных классов* (object class). Объектный класс определяет набор обязательных и необязательных атрибутов для элемента. Например, объектный класс `person` определяет обязательный атрибут `cn` и необязательный атрибут `telephoneNumber`. Очевидно, что объектные классы отличаются от классов Java, но они также поддерживают наследование. Например, `organizationalPerson` представляет собой подкласс класса `person`, содержащий дополнительные атрибуты.

- *Отличительное имя* (distinguished name — DN), например, `uid=jqpublic,ou=people,dc=mycompany,dc=com`. Отличительное имя представляет собой последовательность атрибутов, которые отражают путь, соединяющий текущую запись с корнем дерева. Возможны альтернативные пути, но один из них должен выполнять функции отличительного имени.

Таблица 4.10. Часто используемые атрибуты LDAP

Идентификатор атрибута	Назначение
`dc`	Доменный компонент
`cn`	Общее имя
`sn`	Фамилия
`dn`	Отличительное имя
`o`	Организация
`ou`	Организационная единица
`uid`	Уникальный идентификатор

На рис. 4.9 показан пример дерева каталогов.

Организация дерева каталогов и информация, которую следует поместить в него, может стать предметом отдельного обсуждения. Здесь мы не будем останавливаться на этом вопросы. Будем считать, что организационная схема разработана, и каталог был заполнен согласованными между собой данными.

Настройка LDAP-сервера

Для выполнения программ, приведенных далее в главе, можно использовать разные LDAP-серверы, настроив их должным образом. Ниже перечислены наиболее часто применяемые серверы:

- IBM Tivoli Directory Server
- Microsoft Active Directory
- Novell eDirectory
- OpenLDAP
- Sun Java System Directory Server

Здесь мы приводим краткие инструкции по настройке OpenLDAP (`http://openldap.org`) — свободно распространяемого сервера, ориентированного на работу в средах Linux и Windows; он также встроен в систему Mac OS X. Если вы будете использовать другой сервер каталогов, основные этапы настройки остаются неизменными.

Если вы собираетесь работать с OpenLDAP, перед запуском сервера потребуется отредактировать файл `slapd.conf`. (В системе Linux этот файл хранится в каталоге `/usr/local/etc/openldap` или `/usr/local/etc/openldap`.) В файле `slapd.conf` нужно изменить запись `suffix` так, чтобы она соответствовала используемому набору данных. Эта запись определяет суффикс отличительного имени для сервера и должна выглядеть следующим образом:

```
suffix  "dc=mycompany,dc=com"
```

Вам также следует наделить пользователя LDAP правами администратора, чтобы он мог изменять данные в каталоге.

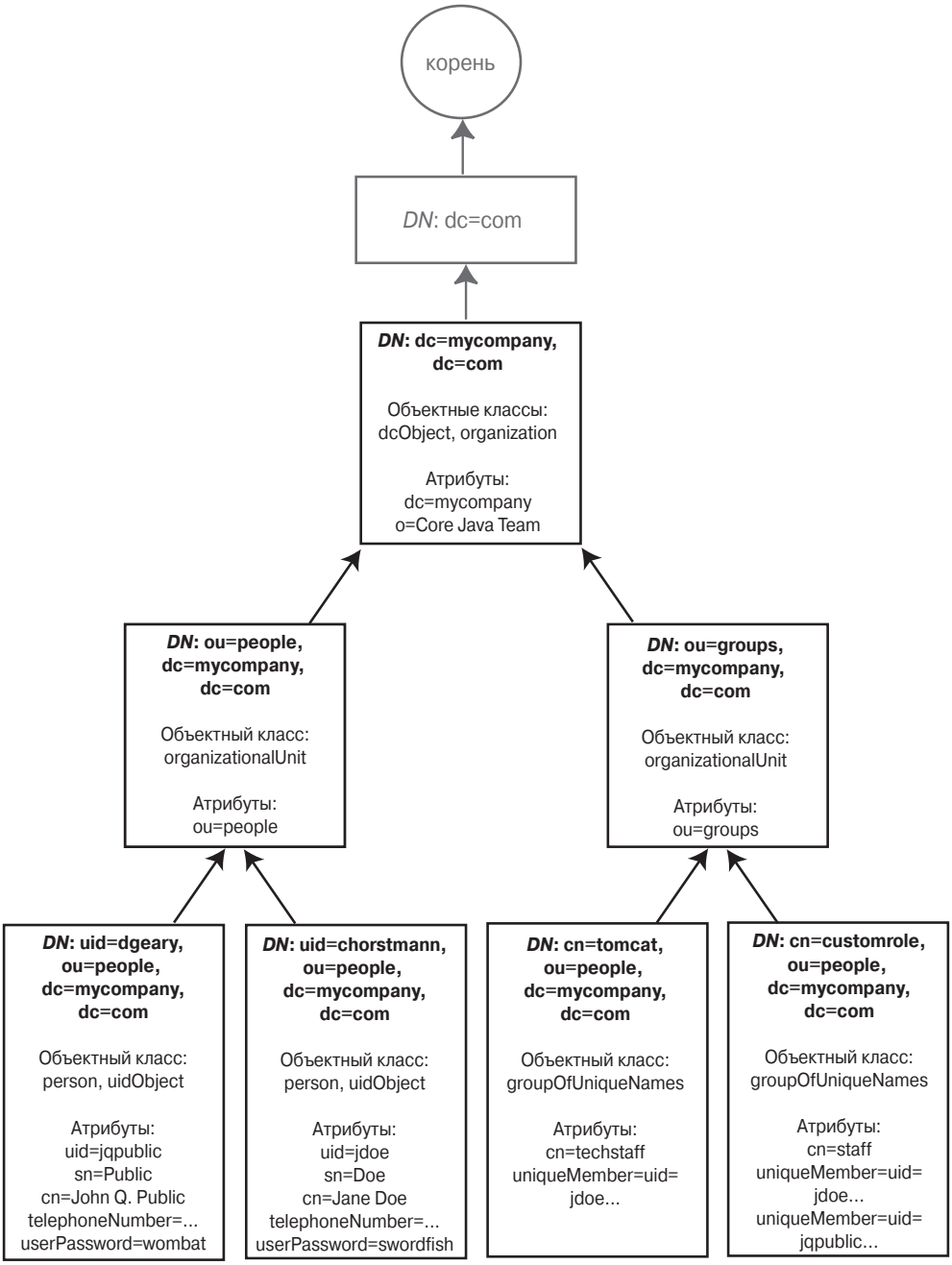


Рис. 4.9. Дерево каталогов

В OpenLDAP для этого необходимо добавить в файл `slapd.conf` следующие строки:

```
rootdn  "cn=Manager,dc=mycompany,dc=com"
rootpw  secret
```

Мы рекомендуем, чтобы вы определили настройки авторизации, хотя они не являются обязательными для выполнения примеров в этом разделе. Следующие настройки в файле `slapd.conf` разрешают пользователю `Manager` читать и записывать пароли, а всем пользователям — читать все остальные атрибуты.

```
access to attr=userPassword
  by dn.base="cn=Manager,dc=mycompany,dc=com" write
  by self write
  by * none
access to *
  by dn.base="cn=Manager,dc=mycompany,dc=com" write
  by self write
  by * read
```

Теперь вы можете запускать LDAP-сервер. В системе Linux для этого следует вызвать службу `slapd` (обычно она находится в каталоге `/usr/sbin` или `/usr/local/libexec`.)

Заполните сервер информацией. Большинство серверов допускают импортирование данных LDIF (Lightweight Directory Interchange Format — формат обмена LDAP). LDIF представляет собой формат, предполагающий перечисление в символьном виде записей каталога, включающих отличительные имена, объектные классы и атрибуты. В листинге 4.5 приведен пример LDIF-файла.

Работая с OpenLDAP, вы можете использовать для включения данных в каталог инструмент `ldapadd`:

```
ldapadd -f sample.ldif -x -D "cn=Manager,dc=mycompany,dc=com" -w secret
```

Листинг 4.5. Содержимое файла `sample.ldif`

```
1.  # Определение записи верхнего уровня.
2.  dn: dc=mycompany,dc=com
3.  objectClass: dcObject
4.  objectClass: organization
5.  dc: mycompany
6.  o: Core Java Team
7.
8.  # Определение записи, содержащей информацию о пользователях.
9.  # Поиск пользователя основывается на этой записи.
10. dn: ou=people,dc=mycompany,dc=com
11. objectClass: organizationalUnit
12. ou: people
13.
14. # Определение записи для пользователя John Q. Public.
15. dn: uid=jqpublic,ou=people,dc=mycompany,dc=com
16. objectClass: person
17. objectClass: uidObject
18. uid: jqpublic
19. sn: Public
20. cn: John Q. Public
21. telephoneNumber: +1 408 555 0017
22. userPassword: wombat
23.
```

```
24.  # Определение записи для пользователя Jane Doe
25.  dn: uid=jdoe,ou=people,dc=mycompany,dc=com
26.  objectClass: person
27.  objectClass: uidObject
28.  uid: jdoe
29.  sn: Doe
30.  cn: Jane Doe
31.  telephoneNumber: +1 408 555 0029
32.  userPassword: heffalump
33.
34.  # Определение записи, содержащей информацию о группах LDAP.
35.  # Поиск ролей основывается на этой записи.
36.  dn: ou=groups,dc=mycompany,dc=com
37.  objectClass: organizationalUnit
38.  ou: groups
39.
40.  # Определение записи для группы "techstaff".
41.  dn: cn=techstaff,ou=groups,dc=mycompany,dc=com
42.  objectClass: groupOfUniqueNames
43.  cn: techstaff
44.  uniqueMember: uid=jdoe,ou=people,dc=mycompany,dc=com
45.
46.  # Определение записи для группы "staff".
47.  dn: cn=staff,ou=groups,dc=mycompany,dc=com
48.  objectClass: groupOfUniqueNames
49.  cn: staff
50.  uniqueMember: uid=jqpublic,ou=people,dc=mycompany,dc=com
51.  uniqueMember: uid=jdoe,ou=people,dc=mycompany,dc=com
```

Перед началом работы желательно убедиться, что каталог содержит требуемые данные. Вероятнее всего, вы воспользуетесь для этого продуктом JXplorer (`http://www.jxplorer.org`) или Jarek Gawor's LDAP Browser/Editor (`http://www-unix.mcs.anl.gov/~gawor/ldap/`). Эти Java-программы позволяют просматривать содержимое LDAP-сервера. Для работы с ними необходимо выполнить ряд установок:

- Хост: `localhost`
- Порт: `389`
- Базовое DN: `dc=mycompany,dc=com`
- Пользовательское DN: `cn=Manager, dc=mycompany, dc=com`
- Пароль: `secret`

Убедитесь, что LDAP-сервер запущен, затем установите соединение. Если вы не допустили ошибку, то увидите дерево каталогов, показанное на рис. 4.10.

Доступ к каталогу LDAP

После заполнения базы данных LDAP установите соединение с ней из Java-программы. Начнем с получения контекста каталога. Для этой цели используется приведенная ниже последовательность операций:

```
Hashtable env = new Hashtable();
env.put(Context.SECURITY_PRINCIPAL, username);
env.put(Context.SECURITY_CREDENTIALS, password);
DirContext initial = new InitialDirContext(env);
DirContext context = (DirContext) initial.lookup("ldap://localhost:389");
```

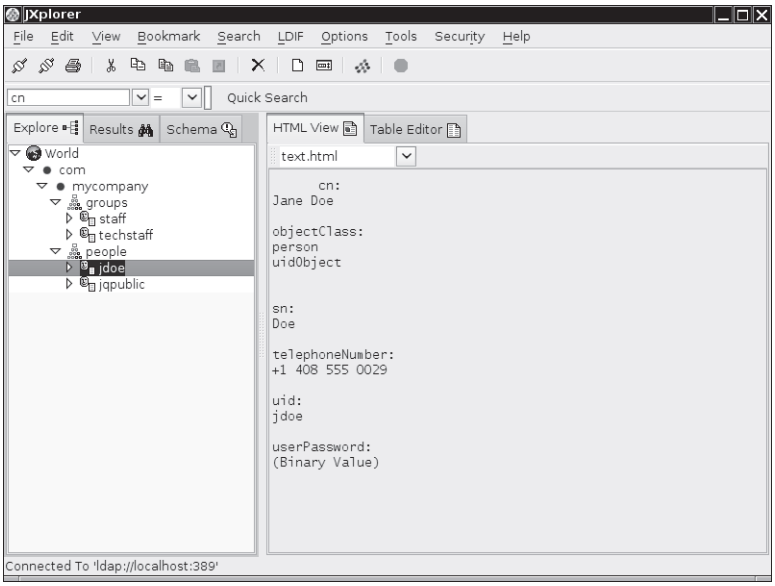


Рис. 4.10. Просмотр дерева каталогов LDAP

Так мы связываемся с LDAP-сервером, который расположен на локальном компьютере. По умолчанию LDAP использует порт с номером 389.

Если при соединении с базой LDAP вы укажете неверное сочетание имени пользователя и пароля, то будет сгенерировано исключение `AuthenticationException`.



В руководстве Sun по работе с JNDI предлагается альтернативный способ подключения к серверу:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://localhost:389");
env.put(Context.SECURITY_PRINCIPAL, userDN);
env.put(Context.SECURITY_CREDENTIALS, password);
DirContext context = new InitialDirContext(env);
```

Однако, на наш взгляд, поступать так не стоит, поскольку при этом в состав кода включается информация о поставщике LDAP. В JNDI реализован специальный механизм конфигурирования поставщиков, и не стоит отказываться от него.

Для просмотра атрибутов определенной записи необходимо сформировать отличительное имя записи и использовать метод `getAttributes()`:

```
Attributes attrs = context.getAttributes(
   "uid=jqpublic,ou=people,dc=mycompany,dc=com");
```

Для получения конкретного атрибута используется метод `get()`:

```
Attribute commonNameAttribute = attrs.get("cn");
```

Для перебора всех атрибутов используется класс `NamingEnumeration`. Разработчики этого класса посчитали возможным внести свой вклад в улучшение стандартного протокола организации итераций и предложили приведенный ниже шаблон:

```
NamingEnumeration<? extends Attribute> attrEnum = attrs.getAll();
while (attrEnum.hasMore())
{
   Attribute attr = attrEnum.next();
   String id = attr.getID();
   . . .
}
```

Обратите внимание на использование метода `hasMore()` вместо `hasNext()`.

Если вы знаете, что атрибут имеет единственное значение, вы можете извлечь его с помощью метода `get()`:

```
String commonName = (String) commonNameAttribute.get();
```

В случае нескольких значений нужно использовать еще один объект `NamingEnumeration`:

```
NamingEnumeration<?> valueEnum = attr.getAll();
while (valueEnum.hasMore())
{
   Object value = valueEnum.next();
   ...
}
```



Начиная с Java SE 5.0, `NamingEnumeration` принадлежит универсальному типу. Выражение `<? extends Attribute>` означает, что перечислению подлежат объекты некоего неизвестного типа, являющегося подтипом `Attribute`. Следовательно, выполнять приведение типов нет необходимости: значения, возвращаемые методом `next()`, имеют тип `Attribute`. Однако `NamingEnumeration<?>` не дает сведений о том, какие объекты перечисляются. Метод `next()` в этом случае возвращает ссылки `Object`.

Теперь вы знаете, как запрашивать информацию из каталога с данными о пользователях. Перейдем к рассмотрению операций, предназначенных для модификации содержимого каталога.

Для того чтобы добавить новую запись, необходимо объединить набор атрибутов в объект `BasicAttributes`. (Класс `BasicAttributes` реализует интерфейс `Attributes`.)

```
Attributes attrs = new BasicAttributes();
attrs.put("uid", "alee");
attrs.put("sn", "Lee");
attrs.put("cn", "Amy Lee");
attrs.put("telephoneNumber", "+1 408 555 0033");
String password = "woozle";
attrs.put("userPassword", password.getBytes());

// Следующий атрибут имеет два значения.
Attribute objclass = new BasicAttribute("objectClass");
objclass.add("uidObject");
objclass.add("person");
attrs.put(objclass);
```

Затем нужно вызвать метод `createSubcontext()`. Данному методу передается отличительное имя новой записи и набор атрибутов:

```
context.createSubcontext(
   "uid=alee,ou=people,dc=mycompany,dc=com", attrs);
```



Собирая атрибуты, помните, что они проверяются по схеме. Не задавайте неизвестные атрибуты и следите за тем, чтобы все атрибуты, предусмотренные объектным классом, присутствовали. Например, если вы не укажете `sn` для `person`, то при выполнении метода `createSubcontext()` возникнет ошибка.

Для удаления записи необходимо вызвать метод `destroySubcontext()`:

```
context.destroySubcontext("uid=alee,ou=people,dc=mycompany,dc=com");
```

И, наконец, вам может потребоваться изменить атрибуты существующей записи. Для этого используется следующий метод:

```
context.modifyAttributes(distinguishedName, flag, attrs);
```

Здесь в качестве второго параметра задается один из следующих флагов:

```
DirContext.ADD_ATTRIBUTE
DirContext.REMOVE_ATTRIBUTE
DirContext.REPLACE_ATTRIBUTE
```

Параметр `attrs` должен содержать набор атрибутов для добавления, удаления или замены.

Конструктор `BasicAttributes(String, Object)` создает набор, состоящий из одного атрибута. Пример использования данного конструктора приведен ниже.

```
context.modifyAttributes("uid=alee,ou=people,dc=mycompany,dc=com",
   DirContext.ADD_ATTRIBUTE,
   new BasicAttributes("title", "CTO"));
context.modifyAttributes("uid=alee,ou=people,dc=mycompany,dc=com",
   DirContext.REMOVE_ATTRIBUTE,
   new BasicAttributes("telephoneNumber", "+1 408 555 0033"));
context.modifyAttributes("uid=alee,ou=people,dc=mycompany,dc=com",
   DirContext.REPLACE_ATTRIBUTE,
   new BasicAttributes("userPassword", password.getBytes()));
```

Закончив работу с контекстом, нужно закрыть его:

```
context.close();
```

Программа, код которой приведен в листинге 4.6, демонстрирует доступ средствами LDAP к иерархической базе данных. Программа позволяет просматривать, модифицировать и удалять информацию из базы, заполненной данными, которые представлены в листинге 4.5.

Если вы отредактируете запись и щелкнете на кнопке Save (Сохранить), внесенные изменения будут сохранены. При изменении содержимого поля редактирования `uid` создастся новая запись. Если запись существует, она будет обновлена. Для удаления записи служит кнопка Delete (Удалить), как показано на рис. 4.11.

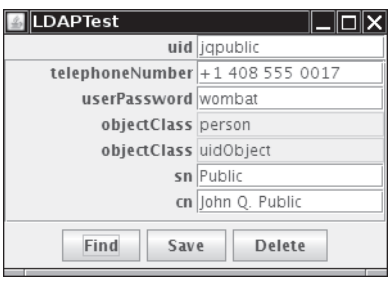


Рис. 4.11. Доступ к иерархической базе данных

Ниже кратно описана работа программы.

- Конфигурационные данные для LDAP-сервера хранятся в файле `ldapserver.properties`. В файле определяются URL, пользовательское имя и пароль для сервера:

  ```
  ldap.username=cn=Manager,dc=mycompany,dc=com
  ldap.password=secret
  ldap.url=ldap://localhost:389
  ```

  Для чтения файла и получения контекста каталога служит метод `getContext()`.
- Когда пользователь щелкает на кнопке Find (Поиск), метод `findEntry()` загружает набор атрибутов для записи с указанным `uid`. Набор атрибутов используется при создании новой панели `DataPanel`.
- Конструктор `DataPanel` перебирает набор атрибутов и создает для каждой пары "идентификатор–значение" метку и поле редактирования.
- Когда пользователь щелкает на кнопке Delete, метод `deleteEntry()` удаляет запись с указанным `uid`.
- По щелчку на кнопке Save `DataPanel` конструктор на основе текущего содержимого полей редактирования создает объект `BasicAttributes`. Метод `saveEntry()` проверяет, изменился ли `uid`. Если пользователь модифицировал `uid`, создается новая запись. В противном случае модифицируется существующая. Код, используемый для модификации, несложен, поскольку мы работаем только с одним атрибутом, допускающим несколько значений, а именно — `objectClass`. В общем случае требуется выполнить необходимую обработку нескольких значений для каждого атрибута.
- Подобно программе, код которой представлен в листинге 4.4, мы закрываем контекст каталога при закрытии фрейма.

Теперь вы имеете сведения об основных операциях, достаточные для решения типичных задач, возникающий при работе с каталогами LDAP. Дополнительную информацию можно найти по адресу `http://java.sun.com/products/jndi/tutorial`.

Листинг 4.6. Содержимое файла `LDAPTest.java`

```
1.    import java.awt.*;
2.    import java.awt.event.*;
3.    import java.io.*;
4.    import java.util.*;
5.    import javax.naming.*;
6.    import javax.naming.directory.*;
7.    import javax.swing.*;
8.
9.    /**
10.    * Эта программа демонстрирует доступ
11.    * к иерархической базе данных LDAP.
12.    * @version 1.01 2007-06-28
13.    * @author Cay Horstmann
14.    */
15.   public class LDAPTest
16.   {
17.      public static void main(String[] args)
18.      {
19.         EventQueue.invokeLater(new Runnable()
20.            {
```

```
21.             public void run()
22.             {
23.                JFrame frame = new LDAPFrame();
24.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25.                frame.setVisible(true);
26.             }
27.          });
28.       }
29.    }
30.    /**
31.     * Фрейм, содержащий панель для отображения данных
32.     * и навигационные кнопки.
33.     */
34.    class LDAPFrame extends JFrame
35.    {
36.       public LDAPFrame()
37.       {
38.          setTitle("LDAPTest");
39.          setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
40.
41.          JPanel northPanel = new JPanel();
42.          northPanel.setLayout(new java.awt.GridLayout(1, 2, 3, 1));
43.          northPanel.add(new JLabel("uid", SwingConstants.RIGHT));
44.          uidField = new JTextField();
45.          northPanel.add(uidField);
46.          add(northPanel, BorderLayout.NORTH);
47.
48.          JPanel buttonPanel = new JPanel();
49.          add(buttonPanel, BorderLayout.SOUTH);
50.
51.          findButton = new JButton("Find");
52.          findButton.addActionListener(new ActionListener()
53.             {
54.                public void actionPerformed(ActionEvent event)
55.                {
56.                   findEntry();
57.                }
58.             });
59.          buttonPanel.add(findButton);
60.
61.          saveButton = new JButton("Save");
62.          saveButton.addActionListener(new ActionListener()
63.             {
64.                public void actionPerformed(ActionEvent event)
65.                {
66.                   saveEntry();
67.                }
68.             });
69.          buttonPanel.add(saveButton);
70.
71.          deleteButton = new JButton("Delete");
72.          deleteButton.addActionListener(new ActionListener()
73.             {
74.                public void actionPerformed(ActionEvent event)
75.                {
76.                   deleteEntry();
77.                }
78.             });
```

```
79.         buttonPanel.add(deleteButton);
80.
81.         addWindowListener(new WindowAdapter()
82.            {
83.               public void windowClosing(WindowEvent event)
84.               {
85.                  try
86.                  {
87.                     if (context != null) context.close();
88.                  }
89.                  catch (NamingException e)
90.                  {
91.                     e.printStackTrace();
92.                  }
93.               }
94.            });
95.      }
96.
97.      /**
98.       * Поиск записи, uid которой указан в поле редактирования.
99.       */
100.     public void findEntry()
101.     {
102.        try
103.        {
104.           if (scrollPane != null) remove(scrollPane);
105.           String dn = "uid=" + uidField.getText() +
106.              ",ou=people,dc=mycompany,dc=com";
107.           if (context == null) context = getContext();
108.           attrs = context.getAttributes(dn);
109.           dataPanel = new DataPanel(attrs);
110.           scrollPane = new JScrollPane(dataPanel);
111.           add(scrollPane, BorderLayout.CENTER);
112.           validate();
113.           uid = uidField.getText();
114.        }
115.        catch (NamingException e)
116.        {
117.           JOptionPane.showMessageDialog(this, e);
118.        }
119.        catch (IOException e)
120.        {
121.           JOptionPane.showMessageDialog(this, e);
122.        }
123.     }
124.
125.     /**
126.      *  Сохранение изменений, произведенных пользователем.
127.      */
128.     public void saveEntry()
129.     {
130.        try
131.        {
132.           if (dataPanel == null) return;
133.           if (context == null) context = getContext();
134.           if (uidField.getText().equals(uid))
135.              // Обновление существующей записи.
```

```
136.          {
137.            String dn = "uid=" + uidField.getText() +
138.                ",ou=people,dc=mycompany,dc=com";
139.            Attributes editedAttrs =
140.              dataPanel.getEditedAttributes();
141.            NamingEnumeration<? extends Attribute> attrEnum =
142.              attrs.getAll();
143.            while (attrEnum.hasMore())
144.            {
145.              Attribute attr = attrEnum.next();
146.              String id = attr.getID();
147.              Object value = attr.get();
148.              Attribute editedAttr = editedAttrs.get(id);
149.              if (editedAttr != null &&
150.                  !attr.get().equals(editedAttr.get()))
151.                context.modifyAttributes(dn,
152.                  DirContext.REPLACE_ATTRIBUTE,
153.                  new BasicAttributes(id, editedAttr.get()));
154.            }
155.          }
156.          else // Создание новой записи.
157.          {
158.            String dn = "uid=" + uidField.getText() +
159.                ",ou=people,dc=mycompany,dc=com";
160.            attrs = dataPanel.getEditedAttributes();
161.            Attribute objclass =
162.              new BasicAttribute("objectClass");
163.            objclass.add("uidObject");
164.            objclass.add("person");
165.            attrs.put(objclass);
166.            attrs.put("uid", uidField.getText());
167.            context.createSubcontext(dn, attrs);
168.          }
169.
170.          findEntry();
171.        }
172.        catch (NamingException e)
173.        {
174.          JOptionPane.showMessageDialog(LDAPFrame.this, e);
175.          e.printStackTrace();
176.        }
177.        catch (IOException e)
178.        {
179.          JOptionPane.showMessageDialog(LDAPFrame.this, e);
180.          e.printStackTrace();
181.        }
182.      }
183.
184.      /**
185.       * Удаление записи, uid которой задан в поле редактирования.
186.       */
187.      public void deleteEntry()
188.      {
189.        try
190.        {
191.          String dn = "uid=" + uidField.getText() +
192.            ",ou=people,dc=mycompany,dc=com";
```

```
193.          if (context == null) context = getContext();
194.          context.destroySubcontext(dn);
195.          uidField.setText("");
196.          remove(scrollPane);
197.          scrollPane = null;
198.          repaint();
199.       }
200.       catch (NamingException e)
201.       {
202.          JOptionPane.showMessageDialog(LDAPFrame.this, e);
203.          e.printStackTrace();
204.       }
205.       catch (IOException e)
206.       {
207.          JOptionPane.showMessageDialog(LDAPFrame.this, e);
208.          e.printStackTrace();
209.       }
210.    }
211.
212.    /**
213.     * Получение контекста. При этом используются свойства,
214.     * заданные в файле ldapserver.properties.
215.     * @return Контекст каталога
216.     */
217.    public static DirContext getContext() throws NamingException, IOException
218.    {
219.       Properties props = new Properties();
220.       FileInputStream in =
221.          new FileInputStream("ldapserver.properties");
222.       props.load(in);
223.       in.close();
224.
225.       String url = props.getProperty("ldap.url");
226.       String username = props.getProperty("ldap.username");
227.       String password = props.getProperty("ldap.password");
228.
229.       Hashtable<String, String> env =
230.          new Hashtable<String, String>();
231.       env.put(Context.SECURITY_PRINCIPAL, username);
232.       env.put(Context.SECURITY_CREDENTIALS, password);
233.       DirContext initial = new InitialDirContext(env);
234.       DirContext context = (DirContext) initial.lookup(url);
235.
236.       return context;
237.    }
238.
239.    public static final int DEFAULT_WIDTH = 300;
240.    public static final int DEFAULT_HEIGHT = 200;
241.
242.    private JButton findButton;
243.    private JButton saveButton;
244.    private JButton deleteButton;
245.
246.    private JTextField uidField;
247.    private DataPanel dataPanel;
248.    private Component scrollPane;
249.
```

```
250.     private DirContext context;
251.     private String uid;
252.     private Attributes attrs;
253.  }
254.
255.  /**
256.   * В данной панели отображается содержимое результирующего набора.
257.   */
258.  class DataPanel extends JPanel
259.  {
260.     /**
261.      * Создание панели данных.
262.      * @param attributes Атрибуты записи
263.      */
264.     public DataPanel(Attributes attrs) throws NamingException
265.     {
266.        setLayout(new java.awt.GridLayout(0, 2, 3, 1));
267.
268.        NamingEnumeration<? extends Attribute> attrEnum = attrs.getAll();
269.        while (attrEnum.hasMore())
270.        {
271.           Attribute attr = attrEnum.next();
272.           String id = attr.getID();
273.
274.           NamingEnumeration<?> valueEnum = attr.getAll();
275.           while (valueEnum.hasMore())
276.           {
277.              Object value = valueEnum.next();
278.              if (id.equals("userPassword")) value = new String((byte[]) value);
279.
280.              JLabel idLabel = new JLabel(id, SwingConstants.RIGHT);
281.              JTextField valueField = new JTextField("" + value);
282.              if (id.equals("objectClass")) valueField.setEditable(false);
283.              if (!id.equals("uid"))
284.              {
285.                 add(idLabel);
286.                 add(valueField);
287.              }
288.           }
289.        }
290.     }
291.
292.     public Attributes getEditedAttributes()
293.     {
294.        Attributes attrs = new BasicAttributes();
295.        for (int i = 0; i < getComponentCount(); i += 2)
296.        {
297.           JLabel idLabel = (JLabel) getComponent(i);
298.           JTextField valueField = (JTextField) getComponent(i + 1);
299.           String id = idLabel.getText();
300.           String value = valueField.getText();
301.           if (id.equals("userPassword")) attrs.put("userPassword", value.getBytes());
302.           else if (!id.equals("") && !id.equals("objectClass")) ttrs.put(id, value);
303.        }
304.        return attrs;
305.     }
306.  }
```

API javax.naming.directory.InitialDirContext 1.3.

- `InitialDirContext(Hashtable env)`

  Создает контекст каталога, используя заданные параметры окружения. Хеш-таблица может содержать информацию, связанную с `Context.SECURITY_PRINCIPAL`, `Context.SECURITY_CREDENTIALS` и другими ключами. Дополнительную информацию по этому вопросу вы найдете в документации по интерфейсу `javax.naming.Context`.

API javax.naming.Context 1.3

- `Object lookup(String name)`

  Выполняет поиск объекта с указанным именем. Возвращаемое значение зависит от контекста. Обычно это поддерево или объект-лист.

- `Context createSubcontext(String name)`

  Создает подконтекст с указанным именем. Подконтекст является дочерним по отношению к текущему контексту. Все компоненты пути, содержащиеся в имени, за исключением последнего, должны существовать.

- `void destroySubcontext(String name)`

  Удаляет подконтекст с указанным именем. Все компоненты пути, содержащиеся в имени, за исключением последнего, должны существовать.

- `void close()`

  Закрывает контекст.

API javax.naming.directory.DirContext 1.3

- `Attributes getAttributes(String name)`

  Возвращает атрибуты записи с указанным именем.

- `void modifyAttributes(String name, int flag, Attributes modes)`

  Модифицирует атрибуты записи с указанным именем. Второй параметр может принимать одно из следующих значений: `DirContext.ADD_ATTRIBUTE`, `DirContext.REMOVE_ATTRIBUTE` или `DirContext.REPLACE_ATTRIBUTE`.

API javax.naming.directory.Attributes 1.3

- `Attribute get(String id)`

  Возвращает атрибут с заданным идентификатором.

- `NamingEnumeration<? extends Attribute> getAll()`

  Формирует нумерованный тип для перебора всех атрибутов в наборе.

- `Attribute put(Attribute attr)`
- `Attribute put(String id, Object value)`

  Добавляет атрибут к набору.

API javax.naming.directory.BasicAttributes 1.3

- `BasicAttributes(String id, Object value)`

  Создает набор атрибутов, содержащий единственный атрибут с заданными идентификатором и значением.

API javax.naming.directory.Attribute 1.3

- `String getID()`

  Возвращает идентификатор атрибута.

- `Object get()`

  Если значения данного атрибута упорядочены, возвращает первое значение. В противном случае возвращает произвольное значение.

- `NamingEnumeration<?> getAll()`

  Формирует перечисление для перебора всех значений данного атрибута.

API javax.naming.NamingEnumeration<T> 1.3

- `boolean hasMore()`

  Возвращает значение `true`, если объект перечисления содержит еще хотя бы один элемент.

- `T next()`

  Возвращает следующий элемент данного перечисления.

В этой главе вы узнали о том, как осуществляется работа с реляционными базами данных в Java; кроме того, здесь вы ознакомились также с иерархическими базами данных. Следующая глава будет посвящена важной теме — интернационализации. В ней будет показано, каким образом можно сделать ваши программы пригодными для работы во всех странах мира.

Глава 5

ИНТЕРНАЦИОНАЛИЗАЦИЯ

В ЭТОЙ ГЛАВЕ...

- Формат вывода чисел
- Дата и время
- Сортировка
- Форматирование сообщений
- Текстовые файлы и наборы символов
- Пакеты ресурсов
- Пример интернационализации программы

Создавая приложение или аплет, разработчик, конечно же, надеется, что его продуктом заинтересуются многие пользователи. К тому же благодаря глобальной сети исчезают барьеры между разными странами. С другой стороны, программист, используя стандартный набор символов ASCII, *сам* создает эти барьеры.

Java — первый язык программирования, в котором изначально были предусмотрены средства интернационализации. Строки формируются из символов Unicode. Поддержка этого стандарта кодирования позволяет создавать Java-программы, обрабатывающие тексты на любом из существующих в мире языков.

Многие программисты считают, что для локализации своих приложений им достаточно использовать Unicode и перевести на нужный язык все сообщения пользовательского интерфейса. Однако этого недостаточно, потому что локализация программы означает нечто большее, чем поддержка Unicode. Дата, время, денежные суммы и даже числа могут по-разному представляться на различных языках. Необходимо найти простой способ настройки команд меню, надписей на кнопках, сообщений и комбинаций клавиш.

В этой главе рассматриваются способы интернационализации Java-приложений и аплетов, а также особенности представления времени, даты, чисел, текста и элементов графического интерфейса пользователя. Кроме того, обсуждаются некоторые инструменты, предназначенные для интернационализации программ. В качестве примера будет рассмотрен аплет, реализующий простой калькулятор, в котором формат представления результатов меняется *в зависимости от географического расположения компьютера, на котором он запускается.*



Дополнительную информацию об интернационализации вы можете найти в документе, расположенном по адресу `http://www.joconner.com/javai18n`, а также на официальном Web-сайте Sun `http://java.sun.com/javase/technologies/core/basic/intl`.

Региональные стандарты

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 точно подметил 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 специального национального алфавита. Существует еще много других более тонких различий, например, в Англии и Германии для представления десятичных чисел применяются разные форматы. Число, понятное англичанам в виде `123,456.78`, должно быть отображено в виде `123.456,78` для пользователей из Германии. Иными словами, точка и запятая в качестве разделителя дробной части и разделителя групп *используются поразному в этих странах*! Похожие различия можно заметить и в способах представления даты. В США привыкли отображать даты в порядке `месяц/день/год`, в Германии используют более практичный порядок — `день/месяц/год`, а в Китае вообще наоборот — `год/месяц/день`. Таким образом, дата `3/22/61` должна быть представлена в виде `22.03.1961` для немецкого пользователя. Конечно, если названия месяцев записаны полностью, то разница в способах представления дат становится очевидной. Например, дата `March 22, 1961` должна быть представлена как `22. März 1961` для немецкого пользователя или как `1961年3月22日` — для китайского.

Существует ряд классов, которые выполняют форматирование, принимая во внимание указанные выше различия. Для управления форматированием используется класс `Locale`. *Региональный стандарт* (locale) определяет язык; кроме того, могут быть указаны географическое расположение и вариант языка.

Например, в США используется следующий региональный стандарт:

language=English, location=United States

В Германии региональный стандарт имеет другой вид:

language=German, location=Germany

В Швейцарии используются четыре официальных языка: немецкий, французский, итальянский и ретороманский. Поэтому немецкие пользователи в Швейцарии, вероятно, захотят использовать следующий региональный стандарт:

language=German, location=Switzerland

В данном случае текст, даты и числа будет форматироваться так же, как и для Германии, но денежные суммы будут отображаться в швейцарских франках, а не в немецких марках.

Если вы зададите только язык, например language=German, то особенности конкретной страны (например, формат представления денежных единиц) не будут учтены.

Вариант языка, используется довольно редко. Например, в настоящее время в норвежском языке (производном от датского) определены два набора правил правописания: традиционный (Bokmål) и новый (Nynorsk). В этом случае для задания традиционных правил орфографии используется параметр, определяющий вариант:

language=Norwegian, location=Norway, variant=Bokmål

Для выражения языка и расположения в компактной и стандартной форме в Java используются коды, определенные Международной организацией по стандартизации (ISO). Язык обозначается двумя строчными буквами в соответствии со стандартом ISO-639-1, а страна (расположение) — двумя прописными буквами в соответствии со стандартом ISO-3166-1. В табл. 5.1 и 5.2 приведены некоторые часто встречающиеся коды.



Полный список обозначений ISO-639-1 можно найти по адресу `http://www.loc.gov/standards/iso639-2/php/code_list.php`, а перечень кодов ISO-3166-1 — по адресу `http://www.iso.org/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/index.html`.

Выбор кодов может показаться случайным, но он, по крайней, мере стандартизирован. Некоторые коды языков происходят от их названий: немецкий = deutsch = `de`, китайский = zhongwen = `zh`.

Чтобы задать региональный стандарт, необходимо объединить код языка, код страны и вариант (если он есть), а затем передать полученную строку в конструктор класса `Locale`.

```
Locale german = new Locale("de");
Locale germanGermany = new Locale("de", "DE");
Locale germanSwitzerland = new Locale("de", "CH");
Locale norwegianNorwayBokmel = new Locale("no", "NO", "B");
```

Таблица 5.1. Языковые коды, определенные стандартом 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
Португальский	pt
Турецкий	tr
Финский	fi
Французский	fr
Шведский	sv
Японский	ja

Таблица 5.2. Коды стран, определенные стандартом 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
Соединенные Штаты Америки	US
Тайвань	TW
Турция	TR
Финляндия	FI
Швейцария	CH
Швеция	SE
Япония	JP

Для удобства пользователей в Java SE предусмотрено несколько предопределенных объектов с региональными настройками:

```
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
```

В Java SE для некоторых языков также имеются объекты, позволяющие указать язык без указания страны:

```
Locale.CHINESE
Locale.ENGLISH
Locale.FRENCH
Locale.GERMAN
Locale.ITALIAN
Locale.JAPANESE
Locale.KOREAN
Locale.SIMPLIFIED_CHINESE
Locale.TRADITIONAL_CHINESE
```

Помимо вызова конструктора или выбора предопределенных объектов, существуют еще два пути получения объектов с региональными настройками.

Статический метод `getDefault()` класса `Locale` позволяет определить региональную настройку, которая используется в операционной системе по умолчанию. Изменить настройку по умолчанию можно, вызвав метод `setDefault()`, однако следует помнить, что этот метод воздействует только на Java-программу, а не на операционную систему в целом. Метод `getLocale()` аплета возвращает региональные настройки того компьютера, на котором он запущен. И, наконец, все зависимые от региональных настроек вспомогательные классы могут возвращать массив поддерживаемых региональных стандартов. Например, приведенный ниже метод возвращает все региональные настройки, поддерживаемые классом `DateFormat`:

```
Locale[] supportedLocales = DateFormat.getAvailableLocales();
```



Программа, рассматриваемая здесь в качестве примера, позволяет переключать региональные настройки, используемые по умолчанию. Запуская программу, укажите язык и регион. Например, в приведенной ниже строке задается немецкий язык для Швейцарии:

```
java -Duser.language=de -Duser.region=CH Program
```

Какие действия можно выполнять на основе полученных региональных настроек? Оказывается, выбор невелик. Единственными полезными методами класса `Locale` являются методы определения кодов языка и страны. Наиболее важным из них является метод `getDisplayName()`, который возвращает строку с описанием региональной настройки. Она содержит не какие-то загадочные двухбуквенные коды, а вполне понятные пользователю слова:

```
German (Switzerland)
```

Но эта строка отображается на используемом по умолчанию языке, что далеко не всегда удобно. Если пользователь выбрал немецкий язык интерфейса, то строку описания следует отобразить именно на немецком языке, для чего можно передать в качестве параметра соответствующую региональную настройку так, как показано ниже:

```
Locale loc = new Locale("de", "CH");
System.out.println(loc.getDisplayName(Locale.GERMAN));
```

В результате выполнения этого кода описание региональной настройки будет выведено на указанном в ней языке:

```
Deutsch (Schweiz)
```

Из данного примера становится ясно, зачем нужны объекты `Locale`. Передавая их методам, способным реагировать на региональные настройки, можно отображать текст на языке, понятном пользователю. Примеры применения этих объектов рассматриваются далее в главе.

API java.util.Locale 1.1

- `Locale(String language)`
- `Locale(String language, String country)`
- `Locale(String language, String country, String variant)`

  Создает объект `Locale` для данных языка, страны и варианта языка.
- `static Locale getDefault()`

  Возвращает используемый по умолчанию региональный стандарт.
- `static void setDefault(Locale loc)`

  Задает используемый по умолчанию региональный стандарт.
- `String getDisplayName()`

  Возвращает строку с описанием текущего регионального стандарта. Эта строка представляется на языке, определяемом текущим региональным стандартом.
- `String getDisplayName(Locale loc)`

  Возвращает строку с описанием текущего регионального стандарта в формате заданного регионального стандарта `loc`.
- `String getLanguage()`

  Возвращает код языка (две строчные буквы) в соответствии со стандартом ISO-639.
- `String getDisplayLanguage()`

  Возвращает название языка в формате текущего регионального стандарта.
- `String getDisplayLanguage(Locale loc)`

  Возвращает название языка в формате заданного регионального стандарта `loc`.
- `String getCountry()`

  Возвращает код страны (две прописные буквы) в соответствии со стандартом ISO-3166.
- `String getDisplayCountry()`

  Возвращает название страны в формате текущего регионального стандарта.
- `String getDisplayCountry(Locale loc)`

  Возвращает название страны в формате заданного регионального стандарта `loc`.

- `String getVariant()`

  Возвращает параметр, определяющий вариант.

- `String getDisplayVariant()`

  Возвращает вариант в формате текущего регионального стандарта.

- `String getDisplayVariant(Locale loc)`

  Возвращает название варианта в формате заданного регионального стандарта `loc`.

- `String toString()`

  Возвращает описание регионального стандарта в виде последовательности кодов языка, страны и варианта, которые разделены символами подчеркивания (например, `"de_CH"`).

API **java.awt.Applet 1.0**

- `Locale getLocale() [1.1]`

  Возвращает объект регионального стандарта для данного аплета.

Форматы чисел

Мы уже упоминали, что в разных странах и регионах используются различные способы представления чисел и денежных сумм. В пакете `java.text` содержатся классы, позволяющие форматировать числа и выполнять разбор их строкового представления. Для форматирования числа в соответствии с конкретным региональным стандартом необходимо выполнить ряд действий.

1. Получить объект регионального стандарта, как было описано в предыдущем разделе.
2. Использовать фабричный метод для получения объекта форматирования.
3. Применить полученный объект форматирования для форматирования числа или разбора его строкового представления.

Здесь в качестве фабричных методов (factory method) используются статические методы `getNumberInstance()`, `getCurrencyInstance()` и `getPercentInstance()` класса `NumberFormat`. Они получают в качестве параметра объект `Locale` и возвращают объекты, предназначенные для форматирования чисел, денежных сумм и значений, выраженных в процентах. Например, для отображения денежной суммы в формате, принятом в Германии, можно использовать 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 кода будет получена следующая строка:

```
123.456,78 €
```

Для обозначения немецкой валюты здесь используется знак €, который располагается в конце строки. Кроме того, обратите внимание на символы, применяемые для обозначения дробной части и разделения десятичных разрядов.

Рассмотрим обратную задачу преобразования строки, записанной в соответствии с определенным региональным стандартом, в число. Для этого предусмотрен метод

parse(), который выполняет синтаксический анализ строки с автоматическим использованием заданного по умолчанию регионального стандарта. В приведенном ниже примере показан способ преобразования строки, введенной пользователем в поле редактирования, в число. Метод parse() способен преобразовывать числа, в которых в качестве разделителя используется точка и запятые.

```
TextField inputField;
...
NumberFormat fmt = NumberFormat.getNumberInstance();
// Получить объект форматирования для используемого
// по умолчанию регионального стандарта
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 исходная строка целое число или число с плавающей точкой. Если это не важно, то для получения числового значения достаточно использовать метод doubleValue() класса Number.



Для объектов типа Number не поддерживается автоматическое приведение к простым типам. Необходимо явным образом вызывать метод doubleValue() или intValue().

Если число представлено в некорректном формате, генерируется исключение ParseException. Например, *не допускается* наличие символов пробела в начале строки, преобразуемой в число. (Для их удаления следует использовать метод trim().) Любые символы, которые располагаются в строке после числа, лишь игнорируются, и исключение в этом случае не возникает.

Очевидно, что классы, возвращаемые методами getXxxInstance(), являются экземплярами не абстрактного класса NumberFormat, а одного из его подклассов. Фабричным методам известно лишь то, как найти объект, представляющий определенный региональный стандарт.

Для получения списка поддерживаемых региональных стандартов можно использовать статический метод getAvailableLocales(), возвращающий массив региональных стандартов, для которых существуют объекты форматирования.

На рис. 5.1 показано окно программы, которая дает возможность поэкспериментировать с объектами форматирования. В верхней части окна расположен список всех региональных стандартов, для которых существуют объекты форматирования. В группе переключателей можно выбрать тип объекта для форматирования чисел, денежных сумм или величин, выраженных в процентах. После выбора нового регионального стандарта и типа объекта форматирования в поле редактирования будет отображено представление заданного числа. Даже просмотрев всего несколько вариантов форматирования чисел и денежных сумм, пользователь, несомненно, составит представление о разнообразии существующих форматов. В поле редактирования можно ввести любое число и щелкнуть на кнопке Parse (Анализ), в результате чего будет вызван метод parse(), выполняющий синтаксический анализ введенной строки. В случае успешного разбора строки результат передается методу format(), а затем отображается на экране. В противном случае в поле редактирования отображается сообщение Parse error.



Рис. 5.1. Окно программы `NumberFormatTest`

Показанный в листинге 5.1 код этой программы имеет довольно простую структуру. Сначала в конструкторе вызывается метод `NumberFormat.getAvailableLocales()`. Затем для каждого поддерживаемого регионального стандарта вызывается метод `getDisplayName()`, а возвращаемые методом результаты помещаются в список. (Строки не сохраняются; решение этой задачи посредством специального раскрывающегося списка описано далее в главе, в разделе “Сортировка”.) Всякий раз, когда пользователь выбирает другой региональный стандарт или щелкает на одном из переключателей, создается новый объект форматирования и обновляется содержимое поля редактирования. Если пользователь щелкает на кнопке Parse, то в таком случае вызывается метод `parse()`, преобразующий строку в число в соответствии с выбранным региональным стандартом.

Листинг 5.1. Содержимое файла `NumberFormatTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.text.*;
4.  import java.util.*;
5.
6.  import javax.swing.*;
7.
8.  /**
9.   * Пример форматирования чисел с учетом
10.  * различных региональных стандартов.
11.  * @version 1.13 2007-07-25
12.  * @author Cay Horstmann
13.  */
14. public class NumberFormatTest
15. {
16.    public static void main(String[] args)
17.    {
18.       EventQueue.invokeLater(new Runnable()
19.          {
20.             public void run()
21.             {
22.                JFrame frame = new NumberFormatFrame();
23.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24.                frame.setVisible(true);
25.             }
26.          });
27.    }
28. }
29. /**
30.  * Фрейм, содержащий группу переключателей для выбора
31.  * формата, список для выбора регионального стандарта,
32.  * поле редактирования для отображения результата форматирования,
33.  * а также кнопку для преобразования строки в число.
34.  */
```

```
35. class NumberFormatFrame extends JFrame
36. {
37.    public NumberFormatFrame()
38.    {
39.       setLayout(new GridBagLayout());
40.
41.       ActionListener listener = new ActionListener()
42.          {
43.       setTitle("NumberFormatTest");
44.             public void actionPerformed(ActionEvent event)
45.             {
46.                updateDisplay();
47.             }
48.          };
49.
50.       JPanel p = new JPanel();
51.       addRadioButton(p, numberRadioButton, rbGroup, listener);
52.       addRadioButton(p, currencyRadioButton, rbGroup, listener);
53.       addRadioButton(p, percentRadioButton, rbGroup, listener);
54.
55.       add(new JLabel("Locale:"),
56.          new GBC(0, 0).setAnchor(GBC.EAST));
57.       add(p, new GBC(1, 1));
58.       add(parseButton, new GBC(0, 2).setInsets(2));
59.       add(localeCombo, new GBC(1, 0).setAnchor(GBC.WEST));
60.       add(numberText, new GBC(1, 2).setFill(GBC.HORIZONTAL));
61.       locales = (Locale[]) NumberFormat.getAvailableLocales().clone();
62.       Arrays.sort(locales, new Comparator<Locale>()
63.          {
64.             public int compare(Locale l1, Locale l2)
65.             {
66.                return l1.getDisplayName().compareTo(l2.getDisplayName());
67.             }
68.          });
69.       for (Locale loc : locales)
70.          localeCombo.addItem(loc.getDisplayName());
71.       localeCombo.setSelectedItem(Locale.getDefault().getDisplayName());
72.       currentNumber = 123456.78;
73.       updateDisplay();
74.
75.       localeCombo.addActionListener(listener);
76.
77.       parseButton.addActionListener(new ActionListener()
78.          {
79.             public void actionPerformed(ActionEvent event)
80.             {
81.                String s = numberText.getText().trim();
82.                try
83.                {
84.                   Number n = currentNumberFormat.parse(s);
85.                   if (n != null)
86.                   {
87.                      currentNumber = n.doubleValue();
88.                      updateDisplay();
89.                   }
90.                   else
91.                   {
```

```
92.                   numberText.setText("Parse error: " + s);
93.                }
94.             }
95.             catch (ParseException e)
96.             {
97.                numberText.setText("Parse error: " + s);
98.             }
99.          }
100.      });
101.     pack();
102.  }
103.
104.  /**
105.   * Включение переключателя в состав контейнера.
106.   * @param p Контейнер, в который помещается переключатель
107.   * @param b Кнопка переключателя
108.   * @param g Группа кнопок
109.   * @param listener Обработчик события, связанного с кнопкой
110.   */
111.  public void addRadioButton(Container p, JRadioButton b, ButtonGroup g,
112.     ActionListener listener)
113.  {
114.     b.setSelected(g.getButtonCount() == 0);
115.     b.addActionListener(listener);
116.     g.add(b);
117.     p.add(b);
118.  }
119.
120.  /**
121.   * Обновление результатов и форматирование числа
122.   * в соответствии с выбором пользователя.
123.   */
124.  public void updateDisplay()
125.  {
126.     Locale currentLocale = locales[localeCombo.getSelectedIndex()];
127.     currentNumberFormat = null;
128.     if (numberRadioButton.isSelected())
129.        currentNumberFormat = NumberFormat.getNumberInstance(currentLocale);
130.     else if (currencyRadioButton.isSelected())
131.        currentNumberFormat = NumberFormat.getCurrencyInstance(currentLocale);
132.     else if (percentRadioButton.isSelected())
133.        currentNumberFormat = NumberFormat.getPercentInstance(currentLocale);
134.     String n = currentNumberFormat.format(currentNumber);
135.     numberText.setText(n);
136.  }
137.
138.  private Locale[] locales;
139.  private double currentNumber;
140.  private JComboBox localeCombo = new JComboBox();
141.  private JButton parseButton = new JButton("Parse");
142.  private JTextField numberText = new JTextField(30);
143.  private JRadioButton numberRadioButton = new JRadioButton("Number");
144.  private JRadioButton currencyRadioButton = new JRadioButton("Currency");
145.  private JRadioButton percentRadioButton = new JRadioButton("Percent");
146.  private ButtonGroup rbGroup = new ButtonGroup();
147.  private NumberFormat currentNumberFormat;
148. }
```

API java.text.NumberFormat 1.1

- `static Locale[] getAvailableLocales()`

  Возвращает массив объектов `Locale`, для которых доступны объекты форматирования `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)`

  Возвращает объект форматирования чисел, денежных сумм или величин, представленных в процентах, для текущего или для заданного регионального стандарта.

- `String format(double x)`
- `String format(long x)`

  Возвращает строку, полученную в результате форматирования заданного числа с плавающей точкой или целого числа.

- `Number parse(String s)`

  Возвращает число, полученное после преобразования строки. Это число может иметь тип `Long` или `Double`. Строка не должна начинаться с пробелов. Любые символы в строке после числа игнорируются. Если преобразование закончилось неудачей, то метод генерирует исключение `ParseException`.

- `void setParseIntegerOnly(boolean b)`
- `boolean isParseIntegerOnly()`

  Устанавливает или возвращает признак того, что данный объект форматирования предназначен для преобразования только целочисленных значений.

- `void setGroupingUsed(boolean b)`
- `boolean isGroupingUsed()`

  Устанавливает или возвращает флаг, указывающий на то, что данный объект форматирования распознает символы разделения групп десятичных разрядов (например, `100,000`).

- `void setMinimumIntegerDigits(int n)`
- `int getMinimumIntegerDigits()`
- `void setMaximumIntegerDigits(int n)`
- `int getMaximumIntegerDigits()`
- `void setMinimumFractionDigits(int n)`
- `int getMinimumFractionDigits()`
- `void setMaximumFractionDigits(int n)`
- `int getMaximumFractionDigits()`

  Устанавливает или возвращает максимальное либо минимальное количество цифр в целой или дробной части числа.

Валюты

Для форматирования денежных сумм используется метод `NumberFormat.getCurrencyInstance()`. Однако этот метод не обеспечивает достаточной гибкости — он возвращает форматированную строку для одной валюты. Предположим, что

вы выписываете счет для американского потребителя, в котором одни суммы представлены в долларах, а другие в евро. Использование двух приведенных ниже объектов форматирования не является решением задачи:

```
NumberFormat dollarFormatter =
   NumberFormat.getCurrencyInstance(Locale.US);
NumberFormat euroFormatter =
   NumberFormat.getCurrencyInstance(Locale.GERMANY);
```

Счет, содержащий такие значения, как $100,000 и 100.000€, будет выглядеть достаточно странно, поскольку при представлении сумм в евро для разделения групп разрядов вместо запятой используется точка.

Для управления форматированием денежных сумм следует использовать класс `Currency`. Для получения объекта `Currency` необходимо передать статическому методу `Currency.getInstance()` идентификатор валюты. Затем нужно вызывать метод `setCurrency()` каждого объекта форматирования. Ниже показано, как настроить объект форматирования евро для американского потребителя:

```
NumberFormat euroFormatter =
   NumberFormat.getCurrencyInstance(Locale.US);
euroFormatter.setCurrency(Currency.getInstance("EUR"));
```

Идентификаторы валют определены стандартом ISO 4217 (`http://www.iso.org/iso/en/prods-services/popstds/currencycodeslist.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`
Индийская рупия	`INR`
Российский рубль	`RUB`

API `java.util.Currency 1.4`

- `static Currency getInstance(String currencyCode)`
- `static Currency getInstance(Locale locale)`

  Возвращает экземпляр класса `Currency`, соответствующий заданному коду ISO 4217 или стране, указанной посредством объекта `Locale`.

- `String toString()`
- `String getCurrencyCode()`

  Возвращает код ISO 4217 для данной валюты.

- `String getSymbol()`
- `String getSymbol(Locale locale)`

  Возвращает символ, обозначающий данную валюту в соответствии с текущими или заданными региональными настройками. Например, в зависимости от объекта `Locale`, доллар США (USD) может обозначаться как `$` или `US$`.

- `int getDefaultFractionDigits()`
  Возвращает число цифр в дробной части для данной валюты, принятое по умолчанию.

Дата и время

При форматировании даты и времени в соответствии с региональными стандартами следует иметь в виду четыре особенности.

- Названия месяцев и дней недели должны быть представлены на местном языке.
- Последовательности указания года, месяца и числа различаются для разных стран и регионов.
- Для отображения дат может использоваться календарь, отличный от григорианского.
- Следует учитывать часовые пояса.

Для учета перечисленных особенностей в Java предусмотрен класс `DateFormat`, который используется почти так же, как и класс `NumberFormat`. Сначала следует сформировать объект регионального стандарта. Для получения массива региональных стандартов, поддерживающих форматирование даты, можно использовать предлагаемый по умолчанию стандарт или вызвать статический метод `getAvailableLocales()`. Далее потребуется вызвать один из трех фабричных методов:

```
fmt = DateFormat.getDateInstance(dateStyle, loc);
fmt = DateFormat.getTimeInstance(timeStyle, loc);
fmt = DateFormat.getDateTimeInstance(dateStyle, timeStyle, loc);
```

Для указания нужного стиля предусмотрен параметр, в качестве которого задается одна из следующих констант:

```
DateFormat.DEFAULT
DateFormat.FULL     (например, Wednesday, September 12, 2007 8:51:03 pm PDT
                     для регионального стандарта США)
DateFormat.LONG     (например, September 12, 2007 8:51:03 pm PDT для регионального стандарта США)
DateFormat.MEDIUM   (например, Sep 12, 2007 8:51:03 pm для регионального стандарта США)
DateFormat.SHORT    (например, 9/12/07 8:51 pm для регионального стандарта США)
```

Фабричный метод возвращает объект форматирования, который можно использовать для форматирования даты:

```
Date now = new Date();
String s = fmt.format(now);
```

Для преобразования строки в дату используется метод `parse()`, который работает аналогично одноименному методу класса `NumberFormat`. Например, приведенный ниже фрагмент кода преобразует строку, введенную пользователем в поле редактирования; при этом учитываются региональные настройки по умолчанию.

```
TextField inputField;
...
DateFormat fmt = DateFormat.getDateInstance(DateFormat.MEDIUM);
Date input = fmt.parse(inputField.getText().trim());
```

К сожалению, пользователь должен вводить дату в конкретном формате. Например, если установлен тип представления даты `MEDIUM` в региональном стандарте США, то предполагается, что введенная дата должна иметь вид `Sep 12, 2007`. Но если пользо-

ватель введет строку `Sep 12 2007` (без запятой) или `9/12/07` (в кратком формате), то это приведет к возникновению исключения `ParseException`.

Для интерпретации неточно указанных дат предусмотрен флаг `lenient`. Если этот флаг установлен, то неверно заданная дата `February 30, 2007` будет автоматически преобразована в дату `March 2, 2007`. Такое поведение вряд ли можно считать безопасным, поэтому данный флаг рекомендуется отключить. Тогда при попытке пользователя ввести некорректное сочетание дня, месяца и года во время преобразования строки в дату будет сгенерировано исключение `IllegalArgumentException`.

В программе, код которой приведен в листинге 5.2, демонстрируется пример использования класса `DateFormat`. С ее помощью можно выбирать различные региональные стандарты 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. Например, так может произойти, если выбрать китайский региональный стандарт.

На рис. 5.2 показано окно программы, которая выполняется на компьютере под управлением китайской версии Windows и успешно отображает все необходимые символы.

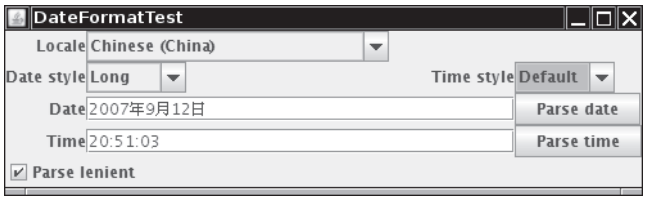


Рис. 5.2. Пример отображения даты и времени при работе с китайской версией Windows

Для проверки преобразования строки в дату введите дату или время в текстовом поле Date (Дата) или Time (Время), установите флажок Parse lenient (Нестрогий анализ), затем щелкните на кнопке Parse date (Анализ даты) или Parse time (Анализ времени).

В данной программе мы используем вспомогательный класс `EnumCombo`. Нам необходимо заполнить раскрывающийся список значениями `Short`, `Medium` и `Long`, а затем организовать автоматическое преобразование выбранного пользователем варианта в целочисленное значение `DateFormat.SHORT`, `DateFormat.MEDIUM` или `DateFormat.LONG`. Вместо того чтобы несколько раз повторять один и тот же код, мы используем отражение. Строка, соответствующая выбранному пользователем пункту, преобразуется в верхний регистр, пробелы заменяются символами подчеркивания, после чего определяется значение статичного поля с этим именем. (Более подробно отражение описано в главе 5 первого тома книги.)



Для вычисления времени с учетом временного пояса следует использовать класс `TimeZone`. Информацию о его применении можно найти в руководстве, расположенном по адресу `http://java.sun.com/developer/JDCTechTips/2003/tt1104.html#2`.

Листинг 5.2. Содержимое файла `DateFormatTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.text.*;
4.  import java.util.*;
5.  import javax.swing.*;
```

```
6.  /**
7.   * Пример форматирования дат для разных региональных стандартов.
8.   * @version 1.13 2007-07-25
9.   * @author Cay Horstmann
10.  */
11. public class DateFormatTest
12. {
13.    public static void main(String[] args)
14.    {
15.       EventQueue.invokeLater(new Runnable()
16.          {
17.             public void run()
18.             {
19.                JFrame frame = new DateFormatFrame();
20.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.                frame.setVisible(true);
22.             }
23.          });
24.    }
25. }
26. /**
27.  * Фрейм, содержащий списки для выбора регионального стандарта,
28.  * типа представления даты и времени, поля редактирования для
29.  * отображения форматированной даты и времени,
30.  * кнопки для преобразования строк, введенных
31.  * в полях редактирования, а также флажок для управления
32.  * режимом автоматического исправления ошибок.
33.  */
34. class DateFormatFrame extends JFrame
35. {
36.    public DateFormatFrame()
37.    {
38.       setTitle("DateFormatTest");
39.
40.       setLayout(new GridBagLayout());
41.       add(new JLabel("Locale"), new GBC(0, 0).setAnchor(GBC.EAST));
42.       add(new JLabel("Date style"), new GBC(0, 1).setAnchor(GBC.EAST));
43.       add(new JLabel("Time style"), new GBC(2, 1).setAnchor(GBC.EAST));
44.       add(new JLabel("Date"), new GBC(0, 2).setAnchor(GBC.EAST));
45.       add(new JLabel("Time"), new GBC(0, 3).setAnchor(GBC.EAST));
46.       add(localeCombo, new GBC(1, 0, 2, 1).setAnchor(GBC.WEST));
47.       add(dateStyleCombo, new GBC(1, 1).setAnchor(GBC.WEST));
48.       add(timeStyleCombo, new GBC(3, 1).setAnchor(GBC.WEST));
49.       add(dateParseButton, new GBC(3, 2).setAnchor(GBC.WEST));
50.       add(timeParseButton, new GBC(3, 3).setAnchor(GBC.WEST));
51.       add(lenientCheckbox, new GBC(0, 4, 2, 1).setAnchor(GBC.WEST));
52.       add(dateText, new GBC(1, 2, 2, 1).setFill(GBC.HORIZONTAL));
53.       add(timeText, new GBC(1, 3, 2, 1).setFill(GBC.HORIZONTAL));
54.
55.       locales = (Locale[]) DateFormat.getAvailableLocales().clone();
56.       Arrays.sort(locales, new Comparator<Locale>()
57.          {
58.             public int compare(Locale l1, Locale l2)
59.             {
60.                return l1.getDisplayName().compareTo(l2.getDisplayName());
61.             }
62.          });
```

```
63.       for (Locale loc : locales)
64.         localeCombo.addItem(loc.getDisplayName());
65.       localeCombo.setSelectedItem(Locale.getDefault().getDisplayName());
66.       currentDate = new Date();
67.       currentTime = new Date();
68.       updateDisplay();
69.
70.       ActionListener listener = new
71.         ActionListener()
72.         {
73.           public void actionPerformed(ActionEvent event)
74.           {
75.             updateDisplay();
76.           }
77.         };
78.
79.       localeCombo.addActionListener(listener);
80.       dateStyleCombo.addActionListener(listener);
81.       timeStyleCombo.addActionListener(listener);
82.
83.       dateParseButton.addActionListener(new ActionListener()
84.         {
85.           public void actionPerformed(ActionEvent event)
86.           {
87.             String d = dateText.getText().trim();
88.             try
89.             {
90.               currentDateFormat.setLenient(
91.                 lenientCheckbox.isSelected());
92.               Date date = currentDateFormat.parse(d);
93.               currentDate = date;
94.               updateDisplay();
95.             }
96.             catch (ParseException e)
97.             {
98.               dateText.setText("Parse error: " + d);
99.             }
100.            catch (IllegalArgumentException e)
101.            {
102.              dateText.setText("Argument error: " + d);
103.            }
104.          }
105.        });
106.
107.      timeParseButton.addActionListener(new ActionListener()
108.        {
109.          public void actionPerformed(ActionEvent event)
110.          {
111.            String t = timeText.getText().trim();
112.            try
113.            {
114.              currentDateFormat.setLenient(
115.                lenientCheckbox.isSelected());
116.              Date date = currentTimeFormat.parse(t);
117.              currentTime = date;
118.              updateDisplay();
119.            }
```

```
120.             catch (ParseException e)
121.             {
122.                timeText.setText("Parse error: " + t);
123.             }
124.             catch (IllegalArgumentException e)
125.             {
126.                timeText.setText("Argument error: " + t);
127.             }
128.          }
129.       });
130.     pack();
131.  }
132.
133.  /**
134.   * Обновление содержимого окна и форматирование даты
135.   * в соответствии с установками, сделанными пользователем.
136.   */
137.  public void updateDisplay()
138.  {
139.     Locale currentLocale = locales[localeCombo.getSelectedIndex()];
140.
141.     int dateStyle = dateStyleCombo.getValue();
142.     currentDateFormat = DateFormat.getDateInstance(dateStyle, currentLocale);
143.     String d = currentDateFormat.format(currentDate);
144.     dateText.setText(d);
145.
146.     int timeStyle = timeStyleCombo.getValue();
147.     currentTimeFormat = DateFormat.getTimeInstance(timeStyle, currentLocale);
148.     String t = currentTimeFormat.format(currentTime);
149.     timeText.setText(t);
150.  }
151.
152.  private Locale[] locales;
153.  private Date currentDate;
154.  private Date currentTime;
155.  private DateFormat currentDateFormat;
156.  private DateFormat currentTimeFormat;
157.  private JComboBox localeCombo = new JComboBox();
158.
159.  private EnumCombo dateStyleCombo = new EnumCombo(DateFormat.class,
160.     new String[] { "Default", "Full", "Long", "Medium", "Short" });
161.
162.  private EnumCombo timeStyleCombo = new EnumCombo(DateFormat.class,
163.     new String[] { "Default", "Full", "Long", "Medium", "Short" });
164.
165.  private JButton dateParseButton = new JButton("Parse date");
166.  private JButton timeParseButton = new JButton("Parse time");
167.
168.  private JTextField dateText = new JTextField(30);
169.  private JTextField timeText = new JTextField(30);
170.  private JTextField parseText = new JTextField(30);
171.
172.  private JCheckBox lenientCheckbox = new JCheckBox("Parse lenient", true);
173. }
```

Листинг 5.3. Содержимое файла `EnumCombo.java`

```
1.  import java.util.*;
2.  import javax.swing.*;
3.
4.  /**
5.   * Раскрывающийся список, позволяющий пользователям
6.   * выбирать значения статических полей.
7.   * @version 1.13 2007-07-25
8.   * @author Cay Horstmann
9.   */
10. public class EnumCombo extends JComboBox
11. {
12.
13.    /**
14.     * Конструктор EnumCombo.
15.     * @param cl Класс
16.     * @param labels Массив имен статических полей класса cl
17.     */
18.    public EnumCombo(Class cl, String[] labels)
19.    {
20.       for (int i = 0; i < labels.length; i++)
21.       {
22.          String label = labels[i];
23.          String name = label.toUpperCase().replace(' ', '_');
24.
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.
39.       setSelectedItem(labels[0]);
40.    }
41.
42.    /**
43.      Данный метод возвращает значение поля,
44.      выбранного пользователем.
45.      @return Значение статического поля
46.    */
47.    public int getValue()
48.    {
49.       return table.get(getSelectedItem());
50.    }
51.
52.    private Map<String, Integer> table = new TreeMap<String, Integer>();
53. }
```

API java.text.DateFormat 1.1

- `static Locale[] getAvailableLocales()`

  Возвращает массив объектов `Locale`, для которых имеются объекты форматирования `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)`

  Возвращает объект форматирования даты, времени либо и даты и времени в формате текущего или заданного регионального стандарта.

  Параметр: `dateStyle`, `timeStyle` Одно из значений `DEFAULT`, `FULL`, `LONG`, `MEDIUM`, `SHORT`

- `String format(Date d)`

  Возвращает строку, полученную в результате форматирования указанных даты и времени.

- `Date parse(String s)`

  Выполняет разбор строки и возвращает заданные в ней дату и время. Строка не должна начинаться с пробелов. Любые символы в строке после даты и времени игнорируются. Если преобразование закончилось неудачно, то метод генерирует исключение `ParseException`.

- `void setLenient(boolean b)`
- `boolean isLenient()`

  Задает или возвращает состояние флага, управляющего нестрогим режимом преобразования строки в дату. Если флаг установлен, то неверно заданная дата, например, `February 30, 1999`, будет автоматически преобразована в дату `March 2, 1999`. По умолчанию нестрогий режим преобразования включен.

- `void setCalendar(Calendar cal)`
- `Calendar getCalendar()`

  Устанавливает или возвращает объект, который поддерживает календарь, используемый для получения из объекта `Date` года, месяцев, дней, часов, минут и секунд. Этот метод используется для замены предлагаемого по умолчанию календаря (обычно григорианского).

- `void setTimeZone(TimeZone tz)`
- `TimeZone getTimeZone()`

  Задает или возвращает часовой пояс, который применяется при форматировании времени. Этот метод используется для замены предлагаемого по умолчанию локального часового пояса. Часовой пояс по умолчанию можно получить, обратившись к операционной системе. Для других региональных настроек часовой пояс определяется на основании географического положения.

- `void setNumberFormat(NumberFormat f)`
- `NumberFormat getNumberFormat()`

  Задает или возвращает формат, используемый для чисел, представляющих годы, месяцы, дни, часы, минуты и секунды.

API java.util.TimeZone 1.1

- `static String[] getAvailableIDs()`

  Возвращает идентификаторы всех поддерживаемых временных поясов.

- `static TimeZone getDefault()`

  Возвращает объект `TimeZone`, представляющий временной пояс по умолчанию для данного компьютера.

- `static TimeZone getTimeZone(String timeZoneId)`

  Возвращает объект `TimeZone`, соответствующий заданному идентификатору часового пояса.

- `String getID()`

  Возвращает идентификатор данного часового пояса.

- `String getDisplayName()`
- `String getDisplayName(Locale locale)`
- `String getDisplayName(boolean daylight, int style)`
- `String getDisplayName(boolean daylight, int style, Locale locale)`

  Возвращает имя часового пояса, соответствующее региональному стандарту, принятому по умолчанию или заданному явно. Если значение параметра `daylight` равно `true`, возвращается имя, в котором учтен перевод на летнее время. Значение параметра `style` может быть равно `SHORT` или `LONG`.

- `boolean useDaylightTime()`

  Возвращает значение `true`, если данный объект `TimeZone` учитывает переход на летнее время.

- `boolean inDaylightTime(Date date)`

  Возвращает значение `true`, если переход на летнее время был выполнен.

Сортировка

Большинству программистов известно, как можно сравнить строки с помощью метода `compareTo()` класса `String`. Результат сравнения `a.compareTo(b)` будет либо отрицательным, если `a` лексикографически меньше `b`, равен нулю, если строки идентичны, либо положительный, если `a` лексикографически больше `b`.

К сожалению, данный метод полезен лишь в случаях, когда слова состоят из прописных английских ASCII-символов. Дело в том, что в методе `compareTo` для сравнения используются порядковые номера символов в кодировке Unicode. Например, строчные буквы в кодировке Unicode имеют б*о*льшие значения, чем прописные, а буквы со знаком ударения имеют еще б*о*льшие значения. Это приводит к абсурдным результатам, например, как в приведенных ниже строках, отсортированных с помощью метода `compareTo`:

```
America
Zulu
able
zebra
Ångström
```

Для сортировки этих слов в том порядке, который используется в словаре, необходимо считать строчные и прописные символы равнозначными. С точки зрения англичанина, правильным будет следующий порядок:

```
able
America
Ángström
zebra
Zulu
```

Однако для шведа указанный порядок будет неправильным. В шведском языке буквы A и Á совершенно разные; более того, Á идет после буквы Z! Поэтому, с точки зрения шведа, эти слова должны быть отсортированы так:

```
able
America
zebra
Zulu
Ángström
```

К счастью, эта проблема решается довольно просто. Сначала нужно получить объект `Locale`, а затем вызвать фабричный метод `getInstance()` для получения объекта-сортировщика `Collator`. Далее сортировка проводится с использованием метода `compare()` объекта `Collator`, а *не* метода `compareTo()` класса `String`:

```
Locale loc = ...;
Collator coll = Collator.getInstance(loc);
if (coll.compare(a, b) < 0) // a располагается перед b ...;
```

Важнее всего то, что класс-сортировщик `Collator` реализует интерфейс `Comparator`. Поэтому для сортировки списка строк можно применять метод `Collections.sort`:

```
Collections.sort(strings, coll);
```

Избирательность сортировки

В сортировщике предусмотрено три уровня *избирательности* (strength): *первостепенный* (primary), *второстепенный* (secondary), *третьестепенный* (tertiary) и *идентичный* (identical). Например, для английского языка различие между символами A и Z учитывается на первостепенном уровне, между A и Á — на второстепенном, а между A и a — на третьестепенном.

Для того чтобы сортировщик обращал внимание только на первостепенные различия, необходимо задать значение `Collator.PRIMARY`. Если указать значение `Collator.SECONDARY`, то в таком случае будут учтены также второстепенные различия. Таким образом, вероятность найти различия в двух строках будет больше при установке большей избирательности. В табл. 5.4 приведены варианты сортировки набора строк для различных уровней избирательности.

Таблица 5.4. Сортировки с разными уровнями избирательности (локализация английская)

`Collator.PRIMARY`	`Collator.SECONDARY`	`Collator.TERTIARY`
Angstrom = Ángström	Angstrom ≠ Ángström	Angstrom ≠ Ángström
Able = able	Able = able	Able ≠ able

Если избирательности будет присвоено `Collator.IDENTICAL`, различия будут отключены. Эта настройка используется главным образом со второй настройкой сортировщика, которая является больше технической — *режимом декомпозиции* (decomposition mode), о котором мы поговорим в следующем разделе.

Декомпозиция

Иногда символ или последовательность символов может быть описана не только посредством Unicode. Например, символ "Å" в кодировке Unicode имеет эквивалент U+00C5; с другой стороны, его можно выразить как простое A (U+0065) с последующим знаком ° ("комбинация с кружочком вверху"; U+030A). Возможно, более удивительным будет то, что последовательность символов "ffi" может быть описана одним символом "Латинская малая лигатура ffi" с кодом U+FB03. (Вообще говоря, это проблема отображения символов, которая не должна решаться на основе применения разных Unicode-символов, но правила установлены не нами.)

Стандарт Unicode определяет четыре *формы нормализации* (D, KD, C и KC) строк. Более подробно об этом можно прочитать по адресу http://www.unicode.org/unicode/reports/tr15/tr15-23.html. Два из них используются для сортировки. В форме нормализации D символы со знаком ударения раскладываются на их базовые буквы и комбинацию с ударениями. Например, Å преобразуется в A и °. В форме нормализации KD раскладываются символы совместимости, такие как лигатура ffi или символ торговой марки ™.

Вы сами выбираете степень нормализации, которая будет использоваться в сортировке. Значение Collator.NO_DECOMPOSITION вообще не нормализует строки. Этот вариант работает быстрее, однако он может быть не пригодным для текста, с помощью которого выражаются символы вомножественных формах. Вариант по умолчанию, Collator.CANONICAL_DECOMPOSITION, использует форму нормализации D. Это самая полезная форма для текста, содержащая ударения, но не лигатуры. И, наконец, "полная декомпозиция" использует форму нормализации KD. Примеры вы можете посмотреть в табл. 5.5.

Таблица 5.5. Различия между режимами декомпозиции

Без декомпозиции	Каноническая декомпозиция	Полная декомпозиция
Å ≠ A°	Å = A°	Å = A°
™ ≠ TM	™ = TM	™ = TM

Если одна строка сравнивается многократно с другими строками, то во избежание ее повторной декомпозиции и повышения эффективности следует сохранить результат декомпозиции в *объекте-ключе сравнения* (collation key). Ниже приводится пример использования метода getCollationKey(), который возвращает объект CollationKey, используемый для более быстрого выполнения всех дальнейших сравнений:

```
String a = . . .;
CollationKey aKey = coll.getCollationKey(a);
if(aKey.compareTo(coll.getCollationKey(b)) == 0)
  // быстрое сравнение
  ...
```

Наконец, вам может понадобиться преобразовать строки в их нормализованные формы, даже если вы не производите сортировку; например, это может быть при сохранении строк в базе данных или соединении с другой программой. В Java SE 6 имеется класс java.text.Normalizer, который занимается процессом нормализации. Например:

```
String name = "Ångström";
String normalized =
 Normalizer.normalize(name, Normalizer.Form.NFD); // используется форма нормализации D
```

Нормализованная строка содержит десять символов. Символы “Å” и “ö” заменяются последовательностями “A°” и “o¨”.

Тем не менее, эта форма обычно не является наилучшей формой для хранения и передачи. Форма нормализации C сначала применяет декомпозицию, а затем комбинирует ударение в стандартизованном порядке. В соответствии с W3C, это рекомендуемый режим передачи данных по сети Internet.

Программа, код которой приведен в листинге 5.4, позволяет экспериментировать с разными видами сортировки. Введите слово в поле редактирования и щелкните на кнопке Add (Добавить) для включения его в список. Список сортируется заново после каждого добавления слова, изменения регионального стандарта, уровня избирательности или режима декомпозиции. Знак равенства (=) указывает на то, что слова считаются идентичными (рис. 5.3).

Локальные имена в комбинированном окне отображаются в отсортированном порядке, посредством сортировщика регионального стандарта, используемого по умолчанию. Если вы запустите эту программу со стандартом US English, вам следует обратить внимание на то, что “Norwegian (Norway,Nynorsk)” будет стоять перед “Norwegian(Norway)”, несмотря даже на то, что значение символа запятой в стандарте Unicode больше, чем значение символа закрывающей скобки.

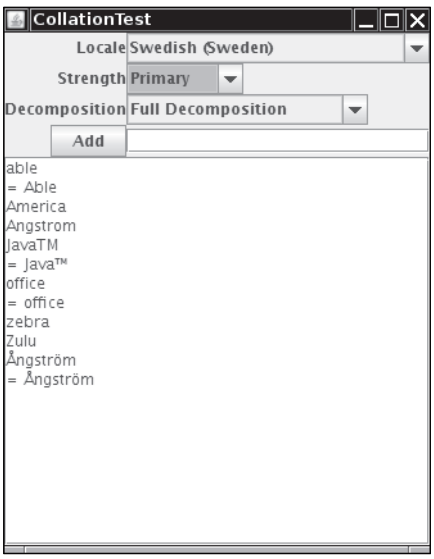


Рис. 5.3. Окно программы `CollationTest`

Листинг 5.4. Содержимое файла `CollationTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.text.*;
4.  import java.util.*;
5.  import java.util.List;
6.
7.  import javax.swing.*;
8.
9.  /**
10.  * Данная программа демонстрирует сортировку строк
11.  * с учетом различных региональных стандартов.
```

```
12.  * @version 1.13 2007-07-25
13.  * @author Cay Horstmann
14.  */
15. public class CollationTest
16. {
17.    public static void main(String[] args)
18.    {
19.       EventQueue.invokeLater(new Runnable()
20.          {
21.             public void run()
22.             {
23.                JFrame frame = new CollationFrame();
24.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25.                frame.setVisible(true);
26.             }
27.          });
28.    }
29. }
30.
31. /**
32.  * Фрейм, содержащий раскрывающиеся списки для выбора
33.  * регионального стандарта, уровня избирательности и правил
34.  * декомпозиции, поле редактирования для ввода слова и кнопку
35.  * для его включения в список, а также текстовую область,
36.  * отображающую результат сортировки слов.
37.  */
38. class CollationFrame extends JFrame
39. {
40.    public CollationFrame()
41.    {
42.       setTitle("CollationTest");
43.
44.       setLayout(new GridBagLayout());
45.       add(new JLabel("Locale"), new GBC(0, 0).setAnchor(GBC.EAST));
46.       add(new JLabel("Strength"), new GBC(0, 1).setAnchor(GBC.EAST));
47.       add(new JLabel("Decomposition"), new GBC(0, 2).setAnchor(GBC.EAST));
48.       add(addButton, new GBC(0, 3).setAnchor(GBC.EAST));
49.       add(localeCombo, new GBC(1, 0).setAnchor(GBC.WEST));
50.       add(strengthCombo, new GBC(1, 1).setAnchor(GBC.WEST));
51.       add(decompositionCombo, new GBC(1, 2).setAnchor(GBC.WEST));
52.       add(newWord, new GBC(1, 3).setFill(GBC.HORIZONTAL));
53.       add(new JScrollPane(sortedWords), new GBC(0, 4, 2, 1).setFill(GBC.BOTH));
54.
55.       locales = (Locale[]) Collator.getAvailableLocales().clone();
56.       Arrays.sort(locales, new Comparator<Locale>()
57.          {
58.             private Collator collator = Collator.getInstance(Locale.getDefault());
59.             public int compare(Locale l1, Locale l2)
60.             {
61.                return collator.compare(l1.getDisplayName(), l2.getDisplayName());
62.             }
63.          });
64.       for (Locale loc : locales)
65.          localeCombo.addItem(loc.getDisplayName());
66.       localeCombo.setSelectedItem(Locale.getDefault().getDisplayName());
67.
68.       strings.add("America");
```

```
      strings.add("able");
      strings.add("Zulu");
      strings.add("zebra");
      strings.add("\u00C5ngstr\u00F6m");
      strings.add("A\u030angstro\u0308m");
      strings.add("Angstrom");
      strings.add("Able");
      strings.add("office");
      strings.add("o\uFB03ce");
      strings.add("Java\u2122");
      strings.add("JavaTM");
      updateDisplay();

      addButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               strings.add(newWord.getText());
               updateDisplay();
            }
         });
      ActionListener listener = new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               updateDisplay();
            }
         };
      localeCombo.addActionListener(listener);
      strengthCombo.addActionListener(listener);
      decompositionCombo.addActionListener(listener);
      pack();
   }
   /**
    * Обновление экранного представления и сортировка строк
    * с учетом установок, сделанных пользователем.
    */
   public void updateDisplay()
   {
      Locale currentLocale =
         (Locale) localeCombo.getSelectedItem();
      localeCombo.setLocale(currentLocale);

      currentCollator = Collator.getInstance(currentLocale);
      currentCollator.setStrength(strengthCombo.getValue());
      currentCollator.setDecomposition(decompositionCombo.getValue());

      Collections.sort(strings, currentCollator);
      sortedWords.setText("");
      for (int i = 0; i < strings.size(); i++)
      {
         String s = strings.get(i);
         if (i > 0 && currentCollator.compare(s, strings.get(i - 1)) == 0)
            sortedWords.append("= ");
         sortedWords.append(s + "\n");
      }
```

```
125.      pack();
126.    }
127.
128.    private List<String> strings = new ArrayList<String>();
129.    private Collator currentCollator;
130.    private Locale[] locales;
131.    private JComboBox localeCombo = new JComboBox();
132.
133.    private EnumCombo strengthCombo = new EnumCombo(Collator.class,
134.       new String[] { "Primary", "Secondary", "Tertiary" });
135.    private EnumCombo decompositionCombo =
136.       new EnumCombo(Collator.class,
137.       new String[] { "Canonical Decomposition",
138.          "Full Decomposition", "No Decomposition" });
139.    private JTextField newWord = new JTextField(20);
140.    private JTextArea sortedWords = new JTextArea(10, 20);
141.    private JButton addButton = new JButton("Add");
142. }
```

API java.text.Collator 1.1

- `static Locale[] getAvailableLocales()`

  Возвращает массив объектов `Locale`, для которых существуют сортировщики `Collator`.

- `static Collator getInstance()`
- `static Collator getInstance(Locale l)`

  Возвращает объект `Collator` для текущего или заданного регионального стандарта.

- `int compare(String a, String b)`

  Возвращает отрицательное значение, если `a` предшествует `b`; значение `0`, если строки считаются идентичными; и положительное значение, если `b` предшествует `a`.

- `boolean equals(String a, String b)`

  Возвращает значение `true`, если строки `a` и `b` считаются идентичными, либо значение `false` в противном случае.

- `void setStrength(int strength)`
- `int getStrength()`

  Устанавливает или возвращает уровень избирательности сортировщика. Чем выше уровень избирательности, тем больше вероятность того, что слова будут признаны различными. Поддерживаются следующие уровни: `Collator.PRIMARY`, `Collator.SECONDARY` и `Collator.TERTIARY`.

- `void setDecomposition(int decomp)`
- `int getDecompositon()`

  Устанавливает или возвращает режим декомпозиции строк. Чем выше степень декомпозиции, тем строже выполняется сравнение строк. Поддерживаются следующие режимы: `Collator.NO_DECOMPOSITION`, `Collator.CANONICAL_DECOMPOSITION` и `Collator.FULL_DECOMPOSITION`.

- `CollationKey getCollationKey(String a)`

  Возвращает ключ с результатами декомпозиции строки, которые используются для быстрого сравнения.

API java.text.CollationKey 1.1

- `int compareTo(CollationKey b)`

  Возвращает отрицательное значение, если данный ключ предшествует ключу `b`; значение `0`, если ключи идентичны; и положительное значение, если данный ключ располагается вслед за ключом `b`.

API java.text.CollationKey 1.1

- `static String normalize(CharSequence str, Normalizer.Form form)`

  Возвращает нормализованную форму `str`. Значение `form` может быть одним из следующих: `ND`, `NKD`, `NC` или `NKC`.

Форматирование сообщений

В библиотеке Java содержится класс `MessageFormat`, который форматирует текст, содержащий фрагменты, представленные посредством переменных:

```
"On {2}, a {0} destroyed {1} houses and caused {3} of damage."
```

Здесь номера в фигурных скобках используются в качестве "заполнителей" для реальных имен и значений. Статический метод `MessageFormat.format()` позволяет подставить значения переменных. В JDK 5.0 поддерживаются методы с переменным числом параметров; таким образом, подстановка может быть выполнена так, как показано ниже:

```
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 рассматриваемом здесь примере переменная `{0}` замещается значением `"hurricane"`, переменная `{1}` заменяется `99` и т.д.



Статический метод `format()` форматирует значения с учетом текущего регионального стандарта. Использовать класс `MessageFormat` с произвольными региональными настройками сложнее, поскольку не существует метода, обеспечивающего работу с переменным числом параметров. Вам нужно поместить значения, которые необходимо отформатировать, в массив `Object[]`, как показано ниже:

```
MessageFormat mf = new MessageFormat(pattern, loc);
String msg = mf.format(new Object[] { значения });
```

В результате обработки строки, рассматриваемой здесь в качестве примера, будет получено следующее предложение:

```
On 1/1/99 12:00 AM, a hurricane destroyed 99 houses and caused 100,000,000 of damage.
```

Однако результат не полностью устраивает нас. Время `12:00 AM` отображать не следует, а сумму ущерба необходимо представить в денежных единицах. Это можно сделать, указав формат для некоторых переменных:

```
"On {2,date,long}, a {0} destroyed {1} houses and caused {3,number,currency} of damage."
```

В результате получим следующую строку:

```
On January 1, 1999, a hurricane destroyed 99 houses and caused $100,000,000 of damage.
```

В составе переменной допускается задавать *тип* и *стиль*. Они разделяются запятыми. Допустимые типы перечислены ниже:

```
number
time
date
choice
```

Если указан тип `number`, то возможны следующие стили:

```
integer
currency
percent
```

Кроме того, в качестве стиля может быть указан шаблон числового формата, например, `$,##0`. (Дополнительную информацию по этому вопросу можно найти в описании класса `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 описании класса `SimpleDateFormat`.)

Форматы выбора (тип `choice`) имеют более сложную структуру и подробно рассматриваются далее в главе.

API java.text.MessageFormat 1.1

- `MessageFormat(String pattern)`
- `MessageFormat(String pattern, Locale loc)`

  Создает объект форматирования сообщения согласно указанному шаблону и региональному стандарту.

- `void applyPattern(String pattern)`

  Задает шаблон для объекта форматирования сообщения.

- `void setLocale(Locale loc)`
- `Locale getLocale()`

  Устанавливает или возвращает региональный стандарт для переменных в составе сообщения. Он используется *только* для последующих шаблонов, заданных с помощью метода `applyPattern()`.

- `static String format(String pattern, Object... args)`

  Форматирует строку согласно шаблону `pattern`, заменяя в нем переменные `{i}` значениями объектов из массива `args[i]`.

- `StringBuffer format(Object args, StringBuffer result, FieldPosition pos)`

  Форматирует шаблон данного объекта `MessageFormat`. Параметр `args` должен представлять собой массив объектов. Форматируемая строка добавляется к значению `result`, которое затем возвращается. Если параметр `pos` эквивалентен `new FieldPosition(MessageFormat.Field.ARGUMENT)`, его свойства `beginIndex` и `endIndex` устанавливаются в соответствии с расположением текста, который замещает переменную `{1}`. Если информация о расположении вас не интересует, в качестве этого параметра следует задать `null`.

API java.text.Format 1.1

- `String format(Object obj)`

  Форматирует заданный объект в соответствии с правилами, определенными посредством текущего объекта форматирования. В процессе работы данный метод обращается к методу `format(obj, new StringBuffer(), new FieldPosition(1)).toString()`.

Форматы выбора

Вернемся к шаблону из предыдущего раздела:

```
"On {2}, a {0} destroyed {1} houses and caused {3} of damage."
```

Если переменную `{0}`, соответствующую стихийному бедствию, заменить значением `earthquake` (землетрясение), то получится предложение с нарушением правил английской грамматики:

```
On January 1, 1999, a earthquake destroyed . . .
```

Для решения этой проблемы в состав значения, замещающего переменную, придется включить артикль `a`:

```
"On {2}, {0} destroyed {1} houses and caused {3} of damage."
```

Теперь переменная `{0}` будет заменяться значениями `a hurricane` или `an earthquake`. Этот способ особенно удобен для перевода сообщений на языки, в которых с каждым родом 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 переменные будут заменяться грамматически правильными сочетаниями артикля и имени существительного, например, `Ein Wirbelsturm` и `Eine Naturkatastrophe`.

Теперь рассмотрим переменную `{1}`. Если стихийное бедствие оказалось не очень разрушительным, то она может быть заменена значением `1`. В таком случае снова получится предложение с нарушением правил английской грамматики:

```
On January 1, 1999, a mudslide destroyed 1 houses and ...
```

Желательно, чтобы сообщение грамотно изменялось в соответствии со значением переменной:

```
no houses
one house
2 houses
...
```

Для решения данной проблемы предусмотрен формат выбора. Согласно данному формату, задается последовательность пар значений, каждая из которых содержит *нижнюю границу* и *строку подстановки*.

Нижняя граница и строка подстановки разделяются символом `#`, а для разделения пар значений используется символ `|`. Ниже приведен пример переменной с указанием формата выбора:

```
{1,choice,0#no houses|1#one house|2#{1} houses}
```

Результаты форматирования, в зависимости от значения `{1}`, показаны в табл. 5.6.

Может возникнуть вопрос, зачем в форматируемой строке дважды указывается переменная {1}? Когда для этой переменной применяется формат выбора и значение оказывается большим или равным 2, возвращается выражение "{1} houses". Оно форматируется снова и включается в результирующую строку.

Таблица 5.6. Строка, отформатированная посредством формата выбора

{1}	Результат
0	"no houses"
1	"one houses"
3	"3 houses"
-1	"no houses"



Данный пример показывает, что разработчики формата выбора приняли не самое лучшее решение. Если есть два варианта форматируемых строк, то для их разделения достаточно двух граничных значений, но согласно формату нужно задать три таких значения. Наименьшее из них никогда не используется.

Синтаксис мог бы быть более понятным, если бы границы указывались между вариантами значений, например следующим образом:

```
no houses|1|one house|2|{1} houses
    // к сожалению, этот формат не поддерживается
```

С помощью символа < можно указать, что предлагаемый вариант должен быть выбран, если нижняя граница строго меньше значения.

Синонимом символа # является символ ≤ (Unicode-код \u2264). При желании вы даже можете указать для первого значения нижнюю границу, равную −∞ (-\u221E):

```
-∞<no houses|0<one house|2≤{1} houses
```

Или, явно указывая коды Unicode:

```
-\u221E<no houses|0<one house|2\u2264{1} houses
```

Завершая разговор о последствиях стихийного бедствия, поместим строку с условиями выбора внутри исходной строки сообщения. В результате получится приведенная ниже конструкция:

```
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.";
```

В немецком варианте она будет выглядеть иначе:

```
String pattern = "{0} zerstцrte am {2,date,long} "
   + "{1,choice,0#kein Haus|1#ein Haus|2#{1} Нduser}"
   + "und richtete einen Schaden von {3,number,currency} an.";
```

Примечательно, что последовательность слов в английском и немецком вариантах разная, но методу format() передается *тот же* массив объектов. Под требуемый порядок слов подстраивается только последовательность появления переменных.

Текстовые файлы и наборы символов

Как вы уже знаете, в языке программирования Java используется кодировка Unicode. Однако во многих операционных системах применяются собственные варианты кодирования символов, часто несовместимые с другими, например ISO 8859-1 (8-битовая кодировка, иногда называемая также ANSI-кодировкой) в США или BIG5 в Тайване.

Сохраняя данные в текстовом файле, необходимо делать это так, чтобы пользователь мог открыть этот файл в другом приложении. Кодировка указывается в конструкторе класса `FileWriter`:

```
out = new FileWriter(filename, "ISO-8859-1");
```

Список поддерживаемых кодировок приведен в главе 12 первого тома.

К сожалению, пока не существует инструментов для организации связи между региональными стандартами и кодировками символов. Например, при выборе стандарта `zh_TW`, принятого в Тайване, в Java не предусмотрен метод, который сообщил бы, что наиболее подходящей кодировкой в этом случае является BIG5.

Кодирование символов в файлах исходного кода

Не следует забывать о том, что программист во время создания программы вынужден *работать с Java-компилятором*. При этом неизбежно использование инструментальных средств, *работающих в локальной операционной системе*. Предположим, что для создания исходных файлов Java-программы используется стандартная программа `Notepad` в китайской версии Windows. Полученный в результате этого исходный код *не является переносимым* (т.е. независимым от платформы) из-за использования локальной кодировки символов (GB или BIG5, в зависимости от типа региональных настроек операционной системы). Классы становятся переносимыми только после компиляции; в этом случае идентификаторы и текстовые сообщения представляются с помощью модифицированной кодировки UTF-8. Сказанное 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.

(Модифицированные форматы UTF-8 и UTF-16 описаны в главе 12 первого тома.)



Указать кодировку исходного файла позволяет опция **`-encoding`**, например:

```
javac -encoding Big5 Myfile.java
```

Для того чтобы сделать исходные файлы переносимыми, следует ограничиться применением только ASCII-символов. Иными словами, необходимо заменить все символы, не принадлежащие ASCII, их эквивалентным представлением в кодировке Unicode. Например, вместо строки `Häuser` следует использовать строку `H\u0084user`. Для преобразования представленных в локальной кодировке текстов в виде ASCII-символов можно применять утилиту `native2ascii`, которая входит в состав JDK. Эта утилита заменяет каждый символ, отличный от ASCII, последовательностью `\uXXXX`, где `XXXX` — четыре шестнадцатеричные цифры Unicode-кода этого символа. При использовании программы `native2ascii` в командной строке нужно указать имя исходного и целевого файла:

```
native2ascii Myfile.java Myfile.temp
```

Для обратного преобразования служит опция -reverse:

```
native2ascii -reverse Myfile.temp Myfile.java
```

Для указания кодировки предусмотрена опция -encoding, за которой следует название кодировки. Допустимые названия кодировок приведены в главе 12 первого тома.

```
native2ascii -encoding Big5 Myfile.java Myfile.temp
```



Мы настоятельно рекомендуем ограничиться использованием в именах классов ASCII-символов. Название *файла класса* происходит от имени соответствующего класса, поэтому особенности обработки символов, отличных от ASCII, в именах зависят от операционной системы. Например, в европейской версии Windows 95 для записи имен файлов используется кодировка *Code Page 437 (original PC)*. Поэтому если вы создадите класс с именем `Bär` и попытаетесь выполнить его в системе Windows 95, то получите сообщение об ошибке `cannot find class BΣr`.

Пакеты ресурсов

При локализации приложений приходится переводить огромное количество сообщений, надписей на кнопках и т.п. Для упрощения задачи рекомендуется собрать строки в отдельном месте, которое называется *ресурсом* (resource). В этом случае переводчику достаточно отредактировать файлы ресурсов, не трогая исходный код программы.

В Java для определения строковых ресурсов используются файлы свойств, а для ресурсов других типов создаются классы ресурсов.



Технология использования ресурсов в Java отличается от технологии использования ресурсов в операционных системах Windows и Macintosh. В выполняемой программе системы Windows такие ресурсы, как меню, диалоговые окна, пиктограммы и сообщения, хранятся отдельно от программы. Поэтому специальный редактор ресурсов позволяет просматривать и модифицировать их без изменения программного кода.



В главе 10 первого тома описывается концепция использования ресурсов, позволяющая размещать файлы данных, аудиофайлы и изображения в JAR-архивах. Метод `getResource()` класса `Class` находит файл, открывает его и возвращает URL, указывающий на ресурс. При размещении файлов в JAR-архивах задачу поиска файлов решает загрузчик классов. Следует отметить, что этот механизм не обеспечивает поддержку региональных стандартов.

Местонахождение пакетов ресурсов

Для локализации приложений создаются так называемые *пакеты ресурсов* (resource bundle). Каждый пакет представляет собой файл свойств или класс, который описывает элементы, специфические для конкретного регионального стандарта (например, сообщения, надписи и т.д.). В каждый пакет помещаются ресурсы для всех региональных стандартов, поддержка которых планируется в программе.

Для именования пакетов ресурсов используются специальные соглашения. Например, ресурсы, специфические для Германии, помещаются в файл с именем *имя_пакета*_de_DE, а ресурсы, общие для стран, в которых используется немецкий язык, размещаются в классе *имя_пакета*_de. Общие правила таковы: ресурсы для конкретной страны именуются по принципу

имя_пакета_язык_страна

Имя файла ресурсов для конкретного языка формируется так:

имя_пакета_язык

Ресурсы, применяемые по умолчанию, помещаются в файл, имя которого не содержит суффикса.

Для загрузки пакета ресурсов используется метод `getBundle()`:

```
ResourceBundle currentResources =
   ResourcesBundle.getBundle("ProgramResources", currentLocale);
```

Метод `getBundle()` пытается загрузить информацию из пакета ресурсов, которая соответствует языку, расположению и варианту текущего регионального стандарта. Если попытка загрузки окончилась неудачей, последовательно отбрасывается вариант, страна и язык. Затем осуществляется поиск ресурса, соответствующего текущему региональному стандарту, и происходит обращение к пакету ресурсов по умолчанию. Если и эта попытка заканчивается неудачей, генерируется исключение `MissingResourceException`.

Таким образом, метод `getBundle()` попытается загрузить следующие пакеты:

имя_пакета_трс_язык_трс_страна_трс_вариант
имя_пакета_трс_язык_трс_страна
имя_пакета_трс_язык
имя_пакета_рсу_язык_рсу_страна_рсу_вариант
имя_пакета_рсу_язык_рсу_страна
имя_пакета_рсу_язык
имя_пакета

(Здесь применены сокращения *трс* – текущий региональный стандарт и *рсу* – региональный стандарт по умолчанию.)

Даже если метод `getBundle()` находит пакет, например *имя_пакета*`_de_DE`, он продолжает искать *имя_пакета*`_de` и *имя_пакета*. Если такие пакеты ресурсов существуют, то они становятся *родительскими* по отношению к пакету *имя_пакета*`_de_DE` в *иерархии ресурсов*. Родительские классы нужны в тех случаях, когда необходимый ресурс не найден в текущем пакете. Другими словами, если нужный ресурс не найден в *имя_пакета*`_de_DE`, проводится поиск в *имя_пакета*`_de` и *имя_пакета*.

Очевидно, что это очень полезный механизм, однако для его реализации вручную программисту пришлось бы выполнить большой объем рутинной работы. Средства поддержки пакетов ресурсов Java автоматически находят ресурсы, наилучшим образом соответствующие конкретному региональному стандарту. Для включения в существующую программу новых локальных настроек необходимо всего лишь дополнить соответствующие пакеты ресурсов.



Создавая приложение, не обязательно помещать все ресурсы в один пакет. Вы можете создать один пакет для надписей на кнопках, другой — для сообщений об ошибках и т.д.

Файлы свойств

Интернационализацию строк осуществить довольно просто. Необходимо поместить все строки в файл свойств, например, `MyProgramStrings.properties`. Файл свойств — это обычный текстовый файл, каждая строка которого содержит ключ и значение.

Пример содержимого такого файла приведен ниже.

```
computeButton=Rechnen
colorName=black
defaultPaperSize=210x297
```

Имя файла свойств выбирается по принципу, описанному в предыдущем разделе:

```
MyProgramStrings.properties
MyProgramStrings_en.properties
MyProgramStrings_de_DE.properties
```

Для загрузки пакета ресурсов из файла свойств применяется приведенный ниже оператор:

```
ResourceBundle bundle =
   ResourceBundle.getBundle("MyProgramStrings", locale);
```

Поиск конкретной строки выполняется следующим образом:

```
String computeButtonLabel = bundle.getString("computeButton");
```



Файлы свойств могут содержать только ASCII-символы. Для размещения в них символов в кодировке Unicode следует использовать формат `\uxxxx`. Например, строка `colorName=Grün` будет иметь вид

```
colorName=Gr\u00FCn
```

Для такого преобразования можно использовать утилиту `native2ascii`.

Классы пакетов

Для поддержки ресурсов, не являющихся строками, нужно определить классы, являющиеся подклассами класса `ResourceBundle`. Выбор имен таких классов осуществляется в соответствии с соглашением об именовании, например:

```
MyProgramResources.java
MyProgramResources_en.java
MyProgramResources_de_DE.java
```

Для загрузки класса используется тот же метод `getBundle()`, что и для загрузки файла свойств:

```
ResourceBundle bundle =
   ResourceBundle.getBundle("MyProgramResources", locale);
```



Если два пакета 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, реализующем пакет ресурсов, поддерживается таблица поиска. Для получения значения используется строка-ключ:

```
Color backgroundColor = (Color) bundle.getObject("backgroundColor");
double[] paperSize = (double[]) bundle.getObject("defaultPaperSize");
```

Самый простой способ реализации класса пакета ресурсов — создание подкласса `ListResourceBundle`. Класс `ListResourceBundle` позволяет помещать все ресурсы в массив объектов и выполняет поиск. Подкласс класса `ListResourceBundle` должен иметь следующую структуру:

```
public class имя_пакета_язык_страна extends ListResourceBundle
{
   public Object[][] getContents() { return contents;  }
   private static final Object[][] contents =
   {
      { ключ1, значение1 },
      { ключ2, значение2 },
      . . .
   }
}
```

Пример классов, созданных на базе `ListResourceBundle`, приведен ниже.

```
public class ProgramResources_de extends ListResourceBundle
{
   public Object[][] getContents() { return contents; }
   private static final Object[][] contents =
   {
      { "backgroundColor", Color.black },
      { "defaultPaperSize", new double[] { 210, 297 } }
   }
}

public class ProgramResources_en_US extends ListResourceBundle
{
   public Object[][] getContents() { return contents; }
   private static final Object[][] contents =
   {
      { "backgroundColor", Color.blue },
      { "defaultPaperSize", new double[] { 216, 279 } }
   }
}
```



Размеры бумаги заданы в миллиметрах. Во всех странах мира, кроме США и Канады, используется бумага, размеры которой определены стандартом ISO 216 (дополнительную информацию по этому вопросу вы найдете в документе `http://www.cl.cam.ac.uk/~mgk25/iso-paper.html`). В соответствии с данными ассоциации мер США (U.S. Metric Association, `http://lamar.colostate.edu/~hillger`) лишь три страны в мире — Либерия, Мьянма (Бирма) и США — не приняли официально метрическую систему.

Класс пакета ресурсов можно также создать как подкласс класса `ResourceBundle`. В этом случае вам придется реализовать два метода, предназначенные для получения объекта `Enumeration`, содержащего ключи, и для извлечения значения, соответствующего конкретному ключу:

```
Enumeration<String> getKeys()
Object handleGetObject(String key)
```

Метод `getObject()` класса `ResourceBundle` вызывает определяемый вами метод `handleGetObject()`.



В версии Java SE 6 вы можете выбирать альтернативные механизмы хранения ваших ресурсов. Например, вы можете настроить механизм загрузки ресурсов для выборки ресурсов из XML-файлов или баз данных. Более подробно об этом см. на странице `http://java.sun.com/developer/technicalArticles/javase/i18n_enhance`.

API java.util.ResourceBundle 1.1

- `static ResourceBundle getBundle(String baseName, Locale loc)`
- `static ResourceBundle getBundle(String baseName)`

  Загружает класс пакета ресурсов с заданным именем, а также его родительские классы для указанного регионального стандарта. Если классы пакетов расположены в Java-пакете, то должно быть указано полное имя, например, `intl.Programresources`. Классы пакетов ресурсов должны быть объявлены общедоступными (`public`), чтобы метод `getBundle()` мог обращаться к ним.

- `Object getObject(String name)`

  Извлекает объект из пакета ресурсов или его родительских пакетов.

- `String getString(String name)`

  Извлекает объект из пакета ресурсов или его родительских пакетов и приводит к типу `String`.

- `String[] getStringArray(String name)`

  Извлекает объект из пакета ресурсов или его родительских пакетов и представляет в виде массива строк.

- `Enumeration<String> getKeys()`

  Возвращает объект `Enumeration`, содержащий ключи текущего пакета ресурсов. При этом в объект `Enumeration` также помещаются ключи из родительских пакетов ресурсов.

- `Object handleGetObject(String key)`

  Если вы реализуете собственный механизм поиска ресурсов, этот метод следует переопределить так, чтобы он возвращал значение, соответствующее указанному ключу.

Пример приложения

В этом разделе изложенный выше материал применяется на практике для интернационализации аплета-калькулятора. Этот калькулятор определяет, достаточно ли вы отчисляете денег для выхода на пенсию. Вы вводите свой возраст, сумму, которую вы отчисляете каждый месяц, и задаете другие данные (рис. 5.4).

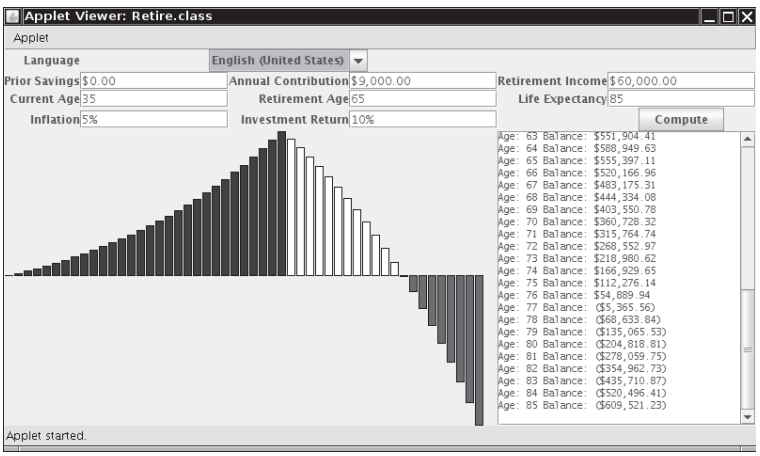


Рис. 5.4. Специализированный калькулятор с английским интерфейсом

В текстовой области и на диаграмме представлены данные о ежегодном состоянии пенсионного счета. Если значения пенсионного счета становятся отрицательными, то в таком случае следует предпринимать какие-то меры, например строже экономить деньги, отложить дату выхода на пенсию и т.п.

Калькулятор имеет три варианта интерфейса: английский, немецкий и китайский. Ниже перечислены основные особенности интернационализации данного аплета.

- Надписи и сообщения переводятся на немецкий и китайский языки. Соответствующие ресурсы находятся в классах `RetireResources_de` и `RetireResources_zh`, а английский вариант интерфейса используется в качестве резервного, и его ресурсы находятся в классе `RetireResources`. Для создания сообщений на китайском языке использовалась программа Notepad в китайской версии Windows, а затем с помощью утилиты `native2ascii` текст был преобразован в Unicode.
- При изменении регионального стандарта заново форматируются надписи и содержимое полей редактирования.
- В полях редактирования числа, денежные суммы и проценты представляются в соответствии с требованиями выбранного регионального стандарта.
- Для форматирования расчетного поля используется класс `MessageFomat`. Строка формата для каждого языка хранится в пакете ресурсов.
- Цвет диаграммы 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 аплета, а в листингах 5.9–5.11 — файлы свойств для локализованных строк. На рис. 5.5 и 5.6 показаны немецкий и китайский варианты интерфейса. Для того чтобы увидеть китайские символы, необходимо запустить аплет в китайской версии Windows или вручную установить китайские шрифты.

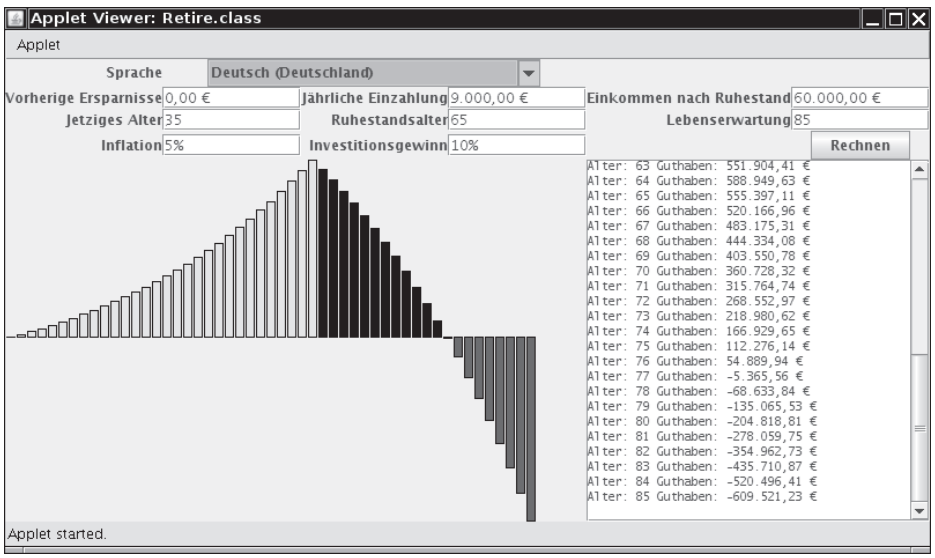


Рис. 5.5. Специализированный калькулятор с немецким интерфейсом

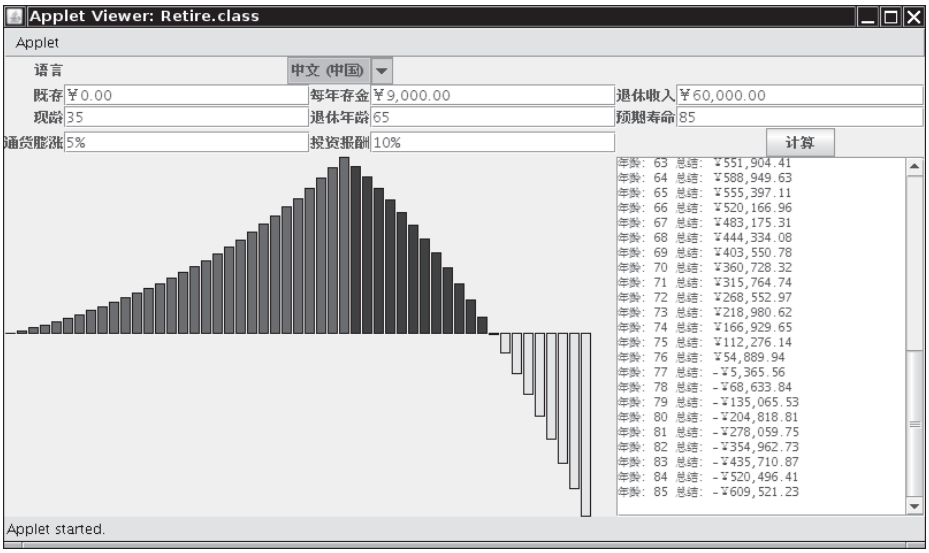


Рис. 5.6. Специализированный калькулятор с китайским интерфейсом

Листинг 5.5. Содержимое файла `Retire.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.awt.geom.*;
4.  import java.util.*;
5.  import java.text.*;
6.  import javax.swing.*;
7.
8.  /**
9.   * Данный аплет реализует калькулятор с английским,
10.  * немецким и китайским интерфейсами.
11.  * @version 1.22 2007-07-25
12.  * @author Cay Horstmann
13.  */
14. public class Retire extends JApplet
15. {
16.    public void init()
17.    {
18.       EventQueue.invokeLater(new Runnable()
19.       {
20.          public void run()
21.          {
22.             initUI();
23.          }
24.       });
25.    }
26.
27.    public void initUI()
28.    {
29.       setLayout(new GridBagLayout());
30.       add(languageLabel, new GBC(0, 0).setAnchor(GBC.EAST));
31.       add(savingsLabel, new GBC(0, 1).setAnchor(GBC.EAST));
```

```
32.       add(contribLabel, new GBC(2, 1).setAnchor(GBC.EAST));
33.       add(incomeLabel, new GBC(4, 1).setAnchor(GBC.EAST));
34.       add(currentAgeLabel, new GBC(0, 2).setAnchor(GBC.EAST));
35.       add(retireAgeLabel, new GBC(2, 2).setAnchor(GBC.EAST));
36.       add(deathAgeLabel, new GBC(4, 2).setAnchor(GBC.EAST));
37.       add(inflationPercentLabel, new GBC(0, 3).setAnchor(GBC.EAST));
38.       add(investPercentLabel, new GBC(2, 3).setAnchor(GBC.EAST));
39.       add(localeCombo, new GBC(1, 0, 3, 1));
40.       add(savingsField, new GBC(1, 1).setWeight(100,
41.         0).setFill(GBC.HORIZONTAL));
42.       add(contribField, new GBC(3, 1).setWeight(100,
43.         0).setFill(GBC.HORIZONTAL));
44.       add(incomeField, new GBC(5, 1).setWeight(100,
45.         0).setFill(GBC.HORIZONTAL));
46.       add(currentAgeField, new GBC(1, 2).setWeight(100,
47.         0).setFill(GBC.HORIZONTAL));
48.       add(retireAgeField, new GBC(3, 2).setWeight(100,
49.         0).setFill(GBC.HORIZONTAL));
50.       add(deathAgeField, new GBC(5, 2).setWeight(100,
51.         0).setFill(GBC.HORIZONTAL));
52.       add(inflationPercentField, new GBC(1, 3).setWeight(100,
53.         0).setFill(GBC.HORIZONTAL));
54.       add(investPercentField, new GBC(3, 3).setWeight(100,
55.         0).setFill(GBC.HORIZONTAL));
56.       add(retireCanvas, new GBC(0, 4, 4, 1).setWeight(0,
57.         100).setFill(GBC.BOTH));
58.       add(new JScrollPane(retireText), new GBC(4, 4, 2,
59.         1).setWeight(0, 100).setFill(GBC.BOTH));
60.
61.       computeButton.setName("computeButton");
62.       computeButton.addActionListener(new ActionListener()
63.          {
64.             public void actionPerformed(ActionEvent event)
65.             {
66.                getInfo();
67.                updateData();
68.                updateGraph();
69.             }
70.          });
71.       add(computeButton, new GBC(5, 3));
72.
73.       retireText.setEditable(false);
74.       retireText.setFont(new Font("Monospaced", Font.PLAIN, 10));
75.
76.       info.setSavings(0);
77.       info.setContrib(9000);
78.       info.setIncome(60000);
79.       info.setCurrentAge(35);
80.       info.setRetireAge(65);
81.       info.setDeathAge(85);
82.       info.setInvestPercent(0.1);
83.       info.setInflationPercent(0.05);
84.
85.       int localeIndex = 0; // По умолчанию выбран региональный стандарт США
86.       for (int i = 0; i < locales.length; i++)
87.          // Выбор регионального стандарта.
88.          if (getLocale().equals(locales[i])) localeIndex = i;
```

```
89.       setCurrentLocale(locales[localeIndex]);
90.
91.       localeCombo.addActionListener(new ActionListener()
92.          {
93.             public void actionPerformed(ActionEvent event)
94.             {
95.                setCurrentLocale((Locale) localeCombo.getSelectedItem());
96.                validate();
97.             }
98.          });
99.    }
100.
101.   /**
102.    * Установка текущего регионального стандарта.
103.    * @param locale Региональный стандарт
104.    */
105.   public void setCurrentLocale(Locale locale)
106.      currentLocale = locale;
107.      localeCombo.setSelectedItem(currentLocale);
108.      localeCombo.setLocale(currentLocale);
109.   {
110.      res = ResourceBundle.getBundle("RetireResources", currentLocale);
111.      resStrings = ResourceBundle.getBundle("RetireStrings", currentLocale);
112.      currencyFmt = NumberFormat.getCurrencyInstance(currentLocale);
113.      numberFmt = NumberFormat.getNumberInstance(currentLocale);
114.      percentFmt = NumberFormat.getPercentInstance(currentLocale);
115.
116.      updateDisplay();
117.      updateInfo();
118.      updateData();
119.      updateGraph();
120.   }
121.
122.   /**
123.    * Обновление всех надписей.
124.    */
125.   public void updateDisplay()
126.   {
127.      languageLabel.setText(resStrings.getString("language"));
128.      savingsLabel.setText(resStrings.getString("savings"));
129.      contribLabel.setText(resStrings.getString("contrib"));
130.      incomeLabel.setText(resStrings.getString("income"));
131.      currentAgeLabel.setText(resStrings.getString("currentAge"));
132.      retireAgeLabel.setText(resStrings.getString("retireAge"));
133.      deathAgeLabel.setText(resStrings.getString("deathAge"));
134.      inflationPercentLabel.setText(resStrings.getString("inflationPercent"));
135.      investPercentLabel.setText(resStrings.getString("investPercent"));
136.      computeButton.setText(resStrings.getString("computeButton"));
137.   }
138.
139.   /**
140.    * Обновление информации в полях редактирования.
141.    */
142.   public void updateInfo()
143.   {
144.      savingsField.setText(currencyFmt.format(info.getSavings()));
145.      contribField.setText(currencyFmt.format(info.getContrib()));
```

```
146.      incomeField.setText(currencyFmt.format(info.getIncome()));
147.      currentAgeField.setText(numberFmt.format(info.getCurrentAge()));
148.      retireAgeField.setText(numberFmt.format(info.getRetireAge()));
149.      deathAgeField.setText(numberFmt.format(info.getDeathAge()));
150.      investPercentField.setText(percentFmt.format(info.getInvestPercent()));
151.      inflationPercentField.setText(percentFmt.format(info.getInflationPercent()));
152.   }
153.
154.   /**
155.    * Обновление даты, отображаемой в текстовой области.
156.    */
157.   public void updateData()
158.   {
159.      retireText.setText("");
160.      MessageFormat retireMsg = new MessageFormat("");
161.      retireMsg.setLocale(currentLocale);
162.      retireMsg.applyPattern(resStrings.getString("retire"));
163.
164.      for (int i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
165.      {
166.         Object[] args = { i, info.getBalance(i) };
167.         retireText.append(retireMsg.format(args) + "\n");
168.      }
169.   }
170.
171.   /**
172.    * Обновление диаграммы.
173.    */
174.   public void updateGraph()
175.   {
176.      retireCanvas.setColorPre((Color) res.getObject("colorPre"));
177.      retireCanvas.setColorGain((Color) res.getObject("colorGain"));
178.      retireCanvas.setColorLoss((Color) res.getObject("colorLoss"));
179.      retireCanvas.setInfo(info);
180.      repaint();
181.   }
182.
183.   /**
184.    * Извлечение из полей редактирования данных,
185.    * введенных пользователем.
186.    */
187.   public void getInfo()
188.   {
189.      try
190.      {
191.         info.setSavings(currencyFmt.parse(
192.            savingsField.getText()).doubleValue());
193.         info.setContrib(currencyFmt.parse(
194.            contribField.getText()).doubleValue());
195.         info.setIncome(currencyFmt.parse(
196.            incomeField.getText()).doubleValue());
197.         info.setCurrentAge(numberFmt.parse(
198.            currentAgeField.getText()).intValue());
199.         info.setRetireAge(numberFmt.parse(
200.            retireAgeField.getText()).intValue());
201.         info.setDeathAge(numberFmt.parse(
202.            deathAgeField.getText()).intValue());
```

```
203.        info.setInvestPercent(percentFmt.parse(
204.          investPercentField.getText()).doubleValue());
205.        info.setInflationPercent(percentFmt.parse(
206.          inflationPercentField.getText()).doubleValue());
207.      }
208.      catch (ParseException e)
209.      {
210.      }
211.    }
212.
213.    private JTextField savingsField = new JTextField(10);
214.    private JTextField contribField = new JTextField(10);
215.    private JTextField incomeField = new JTextField(10);
216.    private JTextField currentAgeField = new JTextField(4);
217.    private JTextField retireAgeField = new JTextField(4);
218.    private JTextField deathAgeField = new JTextField(4);
219.    private JTextField inflationPercentField = new JTextField(6);
220.    private JTextField investPercentField = new JTextField(6);
221.    private JTextArea retireText = new JTextArea(10, 25);
222.    private RetireCanvas retireCanvas = new RetireCanvas();
223.    private JButton computeButton = new JButton();
224.    private JLabel languageLabel = new JLabel();
225.    private JLabel savingsLabel = new JLabel();
226.    private JLabel contribLabel = new JLabel();
227.    private JLabel incomeLabel = new JLabel();
228.    private JLabel currentAgeLabel = new JLabel();
229.    private JLabel retireAgeLabel = new JLabel();
230.    private JLabel deathAgeLabel = new JLabel();
231.    private JLabel inflationPercentLabel = new JLabel();
232.    private JLabel investPercentLabel = new JLabel();
233.
234.    private RetireInfo info = new RetireInfo();
235.
236.    private Locale[] locales = { Locale.US, Locale.CHINA, Locale.GERMANY };
237.    private Locale currentLocale;
238.    private JComboBox localeCombo = new LocaleCombo(locales);
239.    private ResourceBundle res;
240.    private ResourceBundle resStrings;
241.    private NumberFormat currencyFmt;
242.    private NumberFormat numberFmt;
243.    private NumberFormat percentFmt;
244. }
245.
246. /**
247.  * Информация, необходимая для вычисления пенсионного плана.
248.  */
249. class RetireInfo
250. {
251.    /**
252.     * Получение ежегодного состояния счета.
253.     * @param year Год, для которого вычисляется состояние счета
254.     * @return Сумма, доступная (или требуемая) в этом году
255.     */
256.    public double getBalance(int year)
257.    {
258.      if (year < currentAge) return 0;
```

```
259.      else if (year == currentAge)
260.      {
261.         age = year;
262.         balance = savings;
263.         return balance;
264.      }
265.      else if (year == age) return balance;
266.      if (year != age + 1) getBalance(year - 1);
267.      age = year;
268.      if (age < retireAge) balance += contrib;
269.      else balance -= income;
270.      balance = balance * (1 + (investPercent - inflationPercent));
271.      return balance;
272.   }
273.
274.   /**
275.    * Получение суммы предыдущих сбережений.
276.    * @return Объем сбережений
277.    */
278.   public double getSavings()
279.   {
280.      return savings;
281.   }
282.
283.   /**
284.    * Установка суммы предыдущих сбережений.
285.    * @param newValue Объем сбережений
286.    */
287.   public void setSavings(double newValue)
288.   {
289.      savings = newValue;
290.   }
291.
292.   /**
293.    * Получение ежегодной суммы отчислений на пенсионный счет.
294.    * @return Сумма отчислений
295.    */
296.   public double getContrib()
297.   {
298.      return contrib;
299.   }
300.   /**
301.    * Установка ежегодной суммы отчислений на пенсионный счет.
302.    * @param newValue Сумма отчислений
303.    */
304.   public void setContrib(double newValue)
305.   {
306.      contrib = newValue;
307.   }
308.   /**
309.    * Получение информации о ежегодном доходе.
310.    * @return Ежегодный доход
311.    */
312.   public double getIncome()
313.   {
314.      return income;
315.   }
```

```
316.    /**
317.     * Установка величины ежегодного дохода.
318.     * @param newValue Ежегодный доход
319.     */
320.    public void setIncome(double newValue)
321.    {
322.       income = newValue;
323.    }
324.
325.    /**
326.     * Получение текущего возраста.
327.     * @return Текущий возраст
328.     */
329.    public int getCurrentAge()
330.    {
331.       return currentAge;
332.    }
333.
334.    /**
335.     * Установка текущего возраста.
336.     * @param newValue Текущий возраст
337.     */
338.    public void setCurrentAge(int newValue)
339.    {
340.       currentAge = newValue;
341.    }
342.
343.    /**
344.     * Получение возраста выхода на пенсию.
345.     * @return Возраст
346.     */
347.    public int getRetireAge()
348.    {
349.       return retireAge;
350.    }
351.
352.    /**
353.     * Установка возраста выхода на пенсию.
354.     * @param newValue Возраст
355.     */
356.    public void setRetireAge(int newValue)
357.    {
358.       retireAge = newValue;
359.    }
360.
361.    /**
362.     * Получение предполагаемой продолжительности жизни.
363.     * @return Возраст
364.     */
365.    public int getDeathAge()
366.    {
367.       return deathAge;
368.    }
369.
370.    /**
371.     * Установка предполагаемой продолжительности жизни.
372.     * @param newValue Возраст
373.     */
```

```
374.    public void setDeathAge(int newValue)
375.    {
376.       deathAge = newValue;
377.    }
378.
379.    /**
380.     * Получение предполагаемой величины инфляции.
381.     * @return Инфляция в процентах
382.     */
383.    public double getInflationPercent()
384.    {
385.       return inflationPercent;
386.    }
387.
388.    /**
389.     * Установка предполагаемой величины инфляции.
390.     * @param newValue Инфляция в процентах
391.     */
392.    public void setInflationPercent(double newValue)
393.    {
394.       inflationPercent = newValue;
395.    }
396.
397.    /**
398.     * Получение предполагаемого объема вложений.
399.     * @return Объем в процентах
400.     */
401.    public double getInvestPercent()
402.    {
403.       return investPercent;
404.    }
405.
406.    /**
407.     * Установка предполагаемого объема вложений.
408.     * @param newValue Объем в процентах
409.     */
410.    public void setInvestPercent(double newValue)
411.    {
412.       investPercent = newValue;
413.    }
414.
415.    private double savings;
416.    private double contrib;
417.    private double income;
418.    private int currentAge;
419.    private int retireAge;
420.    private int deathAge;
421.    private double inflationPercent;
422.    private double investPercent;
423.
424.    private int age;
425.    private double balance;
426. }
427.
428. /**
429.  * Панель с диаграммой пенсионного плана.
430.  */
```

```
class RetireCanvas extends JPanel
{
   public RetireCanvas()
   {
      setSize(PANEL_WIDTH, PANEL_HEIGHT);
   }

   /**
    * Установка информации, отображаемой на графике.
    * @param newInfo Информация для отображения.
    */
   public void setInfo(RetireInfo newInfo)
   {
      info = newInfo;
      repaint();
   }

   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;
      if (info == null) return;

      double minValue = 0;
      double maxValue = 0;
      int i;
      for (i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
      {
         double v = info.getBalance(i);
         if (minValue > v) minValue = v;
         if (maxValue < v) maxValue = v;
      }
      if (maxValue == minValue) return;

      int barWidth = getWidth() / (info.getDeathAge() -
          info.getCurrentAge() + 1);
      double scale = getHeight() / (maxValue - minValue);

      for (i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
      {
         int x1 = (i - info.getCurrentAge()) * barWidth + 1;
         int y1;
         double v = info.getBalance(i);
         int height;
         int yOrigin = (int) (maxValue * scale);

         if (v >= 0)
         {
            y1 = (int) ((maxValue - v) * scale);
            height = yOrigin - y1;
         }
         else
         {
            y1 = yOrigin;
            height = (int) (-v * scale);
         }

```

```
487.         if (i < info.getRetireAge()) g2.setPaint(colorPre);
488.         else if (v >= 0) g2.setPaint(colorGain);
489.         else g2.setPaint(colorLoss);
490.         Rectangle2D bar = new Rectangle2D.Double(x1, y1, barWidth - 2, height);
491.         g2.fill(bar);
492.         g2.setPaint(Color.black);
493.         g2.draw(bar);
494.       }
495.    }
496.    /**
497.     * Установка цвета диаграммы до даты выхода на пенсию.
498.     * @param color Цвет
499.     */
500.    public void setColorPre(Color color)
501.    {
502.       colorPre = color;
503.       repaint();
504.    }
505.    /**
506.     * Задает цвет диаграммы после даты выхода на пенсию
507.     * при условии положительного баланса.
508.     * @param color Цвет
509.     */
510.    public void setColorGain(Color color)
511.    {
512.       colorGain = color;
513.       repaint();
514.    }
515.    /**
516.     * Задает цвет диаграммы после даты выхода на пенсию
517.     * при условии отрицательного баланса.
518.     * @param color Цвет
519.     */
520.    public void setColorLoss(Color color)
521.    {
522.       colorLoss = color;
523.       repaint();
524.    }
525.    private RetireInfo info = null;
526.    private Color colorPre;
527.    private Color colorGain;
528.    private Color colorLoss;
529.    private static final int PANEL_WIDTH = 400;
530.    private static final int PANEL_HEIGHT = 200;
531. }
```

Листинг 5.6. Содержимое файла `RetireResources.java`

```
1.  import java.awt.*;
2.  /**
3.   * Ресурсы, не являющиеся строками, для
4.   * английского интерфейса калькулятора.
5.   * @version 1.21 2001-08-27
6.   * @author Cay Horstmann
7.   */
8.  public class RetireResources extends java.util.ListResourceBundle
9.  {
```

```
10.    public Object[][] getContents()
11.    {
12.      return contents;
13.    }
14.    static final Object[][] contents =
15.    {
16.     // BEGIN LOCALIZE
17.     { "colorPre", Color.blue }, { "colorGain", Color.white }, { "colorLoss", Color.red }
18.     // END LOCALIZE
19.    };
20.  }
```

Листинг 5.7. Содержимое файла `RetireResources_de.java`

```
1.   import java.awt.*;
2.   /**
3.    * Ресурсы, не являющиеся строками, для
4.    * немецкого интерфейса калькулятора.
5.    * @version 1.21 2001-08-27
6.    * @author Cay Horstmann
7.    */
8.   public class RetireResources_de extends java.util.ListResourceBundle
9.   {
10.    public Object[][] getContents()
11.    {
12.      return contents;
13.    }
14.    static final Object[][] contents =
15.    {
16.      // BEGIN LOCALIZE
17.      { "colorPre", Color.yellow }, { "colorGain", Color.black }, { "colorLoss", Color.red }
18.      // END LOCALIZE
19.    };
20.  }
```

Листинг 5.8. Содержимое файла `RetireResources_zh.java`

```
1.   import java.awt.*;
2.
3.   /**
4.    * Ресурсы, не являющиеся строками, для
5.    * китайского интерфейса калькулятора.
6.    */
7.   public class RetireResources_zh extends java.util.ListResourceBundle
8.   {
9.     public Object[][] getContents()
10.    {
11.      return contents;
12.    }
13.
14.    static final Object[][] contents = {
15.     // BEGIN LOCALIZE
16.     { "colorPre", Color.red }, { "colorGain", Color.blue }, { "colorLoss", Color.yellow }
17.     // END LOCALIZE
18.    };
19.  }
```

Листинг 5.9. Содержимое файла `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 `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 `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}
```

API `java.applet.Applet 1.0`

- `Locale getLocale()` 1.1

  Возвращает текущий региональный стандарт для аплета. Он соответствует текущему региональному стандарту клиентского компьютера, на котором выполняется аплет.

Теперь вам должно быть понятно, как используются средства интернационализации в языке Java. Вы используете пакеты для обеспечения преобразований во множество языков, а средства форматирования и сортировки применяются для обработки текста со специфической локализацией.

В следующей главе будут рассмотрены расширенные вопросы программирования с помощью Swing.

Глава 6

РАСШИРЕННЫЕ СРЕДСТВА SWING

В ЭТОЙ ГЛАВЕ...

- Списки
- Таблицы
- Деревья
- Панель редактирования
- Получение информации о выполнении задачи
- Организаторы компонентов

В этой главе продолжается начатое в первом томе обсуждение набора компонентов Swing, которые предназначены для создания пользовательского интерфейса. В первом томе были представлены только базовые и наиболее распространенные компоненты, хотя пакет Swing содержит много других гораздо более сложных элементов. В этой главе рассматриваются списки, таблицы и деревья, которым посвящена большая часть этой главы. Затем мы перейдем к рассмотрению текстовых компонентов и узнаем больше о простых текстовых полях и областях, которые вам встречались в первом томе. Вы увидите, как добавляются средства проверки и счетчики в текстовые поля, и как отображается структурированный текст, такой как HTML. Затем мы рассмотрим компоненты отображения (индикаторы) хода выполнения длительных операций. В конце главы речь идет об организаторах компонентов в виде панелей с вкладками и панелей рабочего стола с внутренними фреймами.

Списки

Для того чтобы предоставить пользователю возможность выбора среди нескольких значений параметров, можно вместо групп кнопок или флажков (которые занимают довольно много места) использовать обычные или раскрывающиеся списки. Раскрывающиеся списки (combo box) имеют простую структуру и подробно рассматриваются в первом томе. Компонент `JList`, реализующий окно со списком, имеет более сложную структуру, причем при его создании используются некоторые подходы, характерные для формирования деревьев и таблиц. Именно поэтому в данной главе сначала рассматриваются списки.

Чаще всего используются списки, состоящие из строк, но элементами списка могут быть произвольные объекты, допускающие управление своим внешним видом. Компонент `JList` имеет элегантную внутреннюю структуру, благодаря которой и достигается такая универсальность. К сожалению, разработчики компании Sun уделили чрезмерно большое внимание именно элегантности в ущерб простоте использования этого компонента. В некоторых простых ситуациях `JList` не очень удобен в употреблении, поскольку приходится использовать универсальные механизмы. Ниже рассматривается простой и очень распространенный пример организации списка строк, а затем приводится более сложный пример создания списка, который демонстрирует гибкость компонента `JList`.

Компонент `JList`

Этот компонент используется для отображения элементов внутри одного окна. На рис. 6.1 показан очень простой пример списка, в котором пользователь может подобрать характеристики для лисы: `quick` (быстрая), `brown` (коричневая), `hungry` (голодная), `wild` (дикая), а также `static`, `private` и `final` (тут авторы исчерпали известные им характеристики животных и предложили первые пришедшие на ум слова). 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 будет показано в окне программы.

Для создания такого компонента нужно начать с создания массива строк, а затем передать его конструктору компонента `JList`:

```
String[] words = { "quick", "brown", "hungry", "wild", ...};
JList wordList = new JList(words);
```

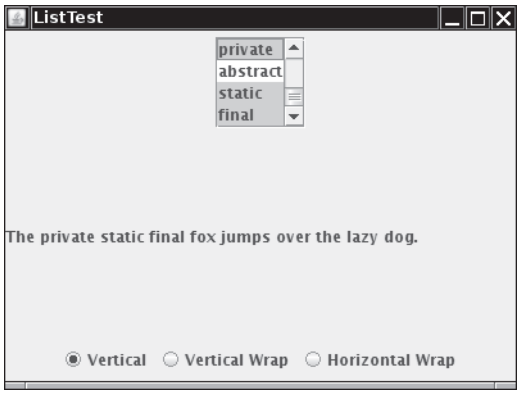


Рис. 6.1. Пример простого списка

Также для этого можно использовать безымянный массив:

```
JList wordList = new JList(new String[]
   { "quick", "brown", "hungry", "wild", ...});
```

В списках не предусмотрены средства автоматической прокрутки, поэтому для организации просмотра длинного списка нужно вставить его в панель с полосой прокрутки `JScrollPane`:

```
JScrollPane scrollPane = new JScrollPane(wordList);
```

А затем нужно вставить эту панель (а не список) в окружающую ее панель.

Следует признать, что разделение функций отображения списка и механизма прокрутки удобно в теоретическом, но не в практическом аспекте. Почти в любом списке приходится прокручивать содержимое. Поэтому совершенно излишне заставлять разработчиков программировать те функции, которые могут быть реализованы по умолчанию, только для того, чтобы можно было оценить универсальность и изящество архитектуры компонента `JList`.

По умолчанию в окне списка отображаются восемь элементов. Но это количество можно изменить с помощью метода `setVisibleRowCount()`, например, так, как показано ниже:

```
wordList.setVisibleRowCount(4); // Отобразить 4 элемента.
```

При организации списка можно выбрать один из трех вариантов расположения пунктов.

- `JList.VERTICAL` (по умолчанию). Все пункты располагаются вертикально друг под другом.
- `JList.VERTICAL_WRAP`. Если количество пунктов превышает заданное число отображаемых строк, формируются новые столбцы (рис. 6.2).
- `JList.HORIZONTAL_WRAP`. Если количество пунктов превышает заданное число отображаемых строк, формируются новые столбцы, однако они заполняются в горизонтальном направлении. Различие между вариантами размещения `JList.VERTICAL_WRAP` и `JList.HORIZONTAL_WRAP` легко понять, сравнив на рис. 6.2 расположение пунктов `quick`, `brown` и `hungry`.

По умолчанию пользователь может выбрать сразу несколько пунктов списка. Например, для выбора несмежных пунктов нужно щелкать кнопкой мыши, удерживая при этом нажатой клавишу <Ctrl>.

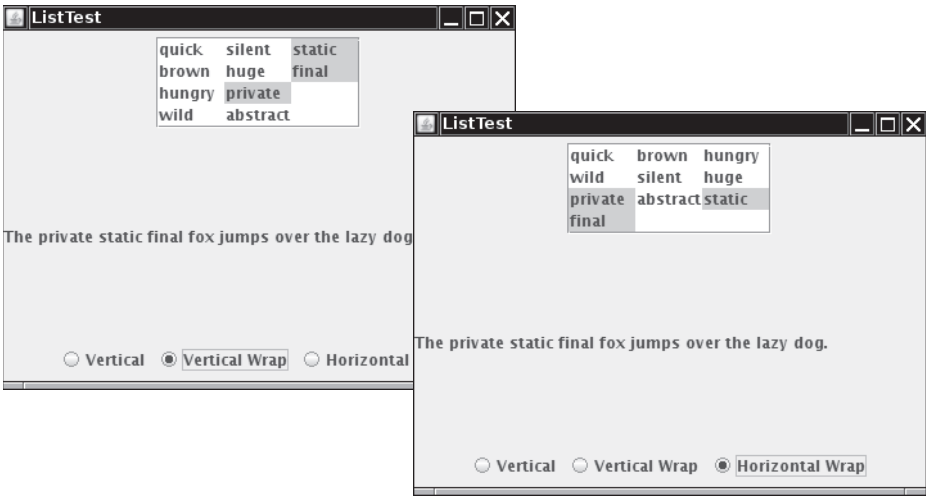


Рис. 6.2. Списки с различным расположением пунктов

Для выбора нескольких смежных пунктов нужно щелкнуть кнопкой мыши сначала на первом из них, а затем, удерживая нажатой клавишу <Shift>, на последнем.

Для ограничения вариантов выбора предусмотрен метод `setSelectionMode()`:

```
wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
   // Разрешается выбирать не более одного пункта списка.

wordList.setSelectionMode

   (ListSelectionModel.SINGLE_INTERVAL_SELECTION);
   // Разрешается выбирать один пункт списка
   // или набор смежных пунктов.
```

Как уже отмечалось в первом томе, базовые компоненты пользовательского интерфейса посылают сообщения о событиях, связанных с действиями пользователя. В списках используется другой механизм уведомления. Вместо события `ActionEvent` при работе со списками перехватывается и обрабатывается событие `ListSelectionEvent`, соответствующее выбору пунктов списка. Для его использования нужно связать компонент с классом-обработчиком и реализовать в последнем метод

```
public void valueChanged(ListSelectionEvent evt);
```

В процессе выбора пользователем пунктов списка генерируется несколько событий. Предположим, что пользователь щелкнул на пункте списка. При нажатии кнопки мыши передается событие об изменении выбора. Это промежуточное событие. Приведенный ниже метод вернет значение `true`, если выбор не был окончательным:

```
event.isAdjusting()
```

Затем при отпускании кнопки мыши генерируется еще одно событие; на этот раз метод `isAdjusting()` возвратит значение `false`. Если обработка промежуточных событий не нужна, то можно отслеживать только те события, для которых метод `isAdjusting()` возвращает значение `false`. Однако для учета всех манипуляций мышью потребуется обрабатывать все события.

После уведомления о событии потребуется определить, какие пункты были выбраны. Для этого служит метод `getSelectedValues()`, который возвращает

массив объектов со всеми выбранными пунктами списка. Далее для *всех* элементов списка необходимо выполнить приведение к типу `String`:

```
Object[] values = list.getSelectedValues();
for (Object value : values)
   какие-то действия с (String) value;
```



Нельзя приводить к типу `String[]` значение типа `Object[]`, возвращаемое методом `getSelectedValues()`. Дело в том, что возвращаемое значение представляет собой массив объектов, а не строк. Поэтому приведение типов придется выполнить отдельно для каждого элемента. Если же вам нужен массив строк, его можно сформировать следующим образом:

```
int length = values.length;
String[] words = new String[length];
System.arrayCopy(values, 0, words, 0, length);
```

Если в списке не допускается множественный выбор, то в таком случае следует применять метод `getSelectedValue()`, который возвращает первый (и единственный) выбранный пункт списка:

```
String value = (String) list.getSelectedValue();
```



В компоненте `JList` не предусмотрена обработка двойного щелчка кнопкой мыши. Разработчики библиотеки Swing предполагали, что список используется только для выбора элементов, а для выполнения какого-либо действия следует нажать соответствующую кнопку. Однако в некоторых реализациях пользовательского интерфейса двойной щелчок кнопкой мыши рассматривается одновременно как выбор элемента и команда на выполнение соответствующего действия. Если вы хотите предоставить пользователю такую возможность, потребуется связать со списком обработчик событий мыши, а затем обрабатывать `MouseEvent`.

```
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 полный код программы, иллюстрирующий использование списка, пунктами которого являются строки. Обратите внимание на метод `valueChanged()`, который создает строку сообщения на основе выбранных пунктов.

Листинг 6.1. Содержимое файла `ListTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import javax.swing.*;
4.  import javax.swing.event.*;
5.
6.  /**
7.   * Пример списка с фиксированным набором строк.
```

```
8.    * @version 1.23 2007-08-01
9.    * @author Cay Horstmann
10.   */
11.  public class ListTest
12.  {
13.    public static void main(String[] args)
14.    {
15.         EventQueue.invokeLater(new Runnable()
16.              {
17.                    public void run()
18.                    {
19.                        JFrame frame = new ListFrame();
20.                        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.                        frame.setVisible(true);
22.                     }
23.               });
24.     }
25.  }
26.  /**
27.    * Этот фрейм содержит список слов и метку, которая отображает
28.    * предложение, составленное из выбранных слов. Учтите, что с
29.    * помощью клавиш <Ctrl> и <Shift> можно выбрать сразу
30.    * несколько слов.
31.    */
32.  class ListFrame extends JFrame
33.  {
34.    public ListFrame()
35.    {
36.       setTitle("ListTest");
37.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
38.
39.       String[] words = { "quick","brown","hungry","wild","silent",
40.                          "huge","private", "abstract","static","final" };
41.
42.       wordList = new JList(words);
43.       wordList.setVisibleRowCount(4);
44.       JScrollPane scrollPane = new JScrollPane(wordList);
45.
46.       listPanel = new JPanel();
47.       listPanel.add(scrollPane);
48.       wordList.addListSelectionListener(new ListSelectionListener()
49.         {
50.           public void valueChanged(ListSelectionEvent event)
51.           {
52.             Object[] values = wordList.getSelectedValues();
53.
54.             StringBuilder text = new StringBuilder(prefix);
55.             for (int i = 0; i < values.length; i++)
56.             {
57.               String word = (String) values[i];
58.               text.append(word);
59.               text.append(" ");
60.             }
61.             text.append(suffix);
62.             label.setText(text.toString());
63.           }
64.         });
```

```
65.       buttonPanel = new JPanel();
66.       group = new ButtonGroup();
67.       makeButton("Vertical", JList.VERTICAL);
68.       makeButton("Vertical Wrap", JList.VERTICAL_WRAP);
69.       makeButton("Horizontal Wrap", JList.HORIZONTAL_WRAP);
70.
71.       add(listPanel, BorderLayout.NORTH);
72.       label = new JLabel(prefix + suffix);
73.       add(label, BorderLayout.CENTER);
74.       add(buttonPanel, BorderLayout.SOUTH);
75.    }
76.
77.    /**
78.      * Создание переключателя для выбора требуемого
79.      * расположения пунктов списка.
80.      * @param label Метка переключателя
81.      * @param orientation Вариант расположения пунктов списка
82.      */
83.    private void makeButton(String label, final int orientation)
84.    {
85.       JRadioButton button = new JRadioButton(label);
86.       buttonPanel.add(button);
87.       if (group.getButtonCount() == 0) button.setSelected(true);
88.       group.add(button);
89.       button.addActionListener(new ActionListener()
90.          {
91.             public void actionPerformed(ActionEvent event)
92.             {
93.                wordList.setLayoutOrientation(orientation);
94.                listPanel.revalidate();
95.             }
96.          });
97.    }
98.
99.    private static final int DEFAULT_WIDTH = 400;
100.   private static final int DEFAULT_HEIGHT = 300;
101.   private JPanel listPanel;
102.   private JList wordList;
103.   private JLabel label;
104.   private JPanel buttonPanel;
105.   private ButtonGroup group;
106.   private String prefix = "The ";
107.   private String suffix = "fox jumps over the lazy dog.";
108. }
```

API javax.swing.JList 1.2

- `JList(Object[] items)`

  Создает список, составленный из указанных пунктов.

- `int getVisibleRowCount()`
- `void setVisibleRowCount(int c)`

  Возвращает или задает количество пунктов списка, которые будут видны без использования полосы прокрутки.

- `int getLayoutOrientation()` 1.4
- `void setLayoutOrientation(int orientation)` 1.4

  Возвращает или устанавливает вариант расположения пунктов списка.

  Параметр: `orientation` Значение `VERTICAL`, `VERTICAL_WRAP` или `HORIZONTAL_WRAP`
- `int getSelectionMode()`
- `void setSelectionMode(int mode)`

  Возвращает или устанавливает режим, определяющий возможность множественного выбора пунктов списка.

  Параметр: `mode` Значение `SINGLE_SELECTION` (только один элемент), `SINGLE_INTERVAL_SELECTION` (только один интервал элементов), `MULTIPLE_INTERVAL_SELECTION` (несколько интервалов элементов).
- `void addListSelectionListener(ListSelectionListener listener)`

  Связывает со списком обработчик, который уведомляет о каждом изменении выбора.
- `Object[] getSelectedValues()`

  Возвращает выбранные значения или пустой массив, если пользователь не выбрал ни одного пункта.
- `Object getSelectedValue()`

  Возвращает первое выбранное значение или значение `null`, пользователь не выбрал ни одного пункта.

API `javax.swing.event.ListSelectionListener 1.2`

- `void valueChanged(ListSelectionEvent e)`

  Вызывается при каждом изменении выбора пунктов списка.

Модели списков

В предыдущем разделе описан наиболее распространенный способ использования списков.

1. Указание фиксированного набора строк для отображения в списке.
2. Помещение списка в состав панели, снабженной полосой прокрутки.
3. Обработка событий, связанных с выбором элементов списка.

Далее рассматриваются более сложные случаи.

- очень длинные списки;
- списки с изменяющимся содержимым;
- списки, которые содержат пункты, отличные от строк.

В предыдущем примере рассматривался компонент `JList` с фиксированным набором строк. Однако набор пунктов списка может изменяться. Как можно было бы организовать вставку и удаление элементов списка? Удивительно, но методов подобного рода в классе `JList` не существует. Рассмотрим более подробно внутреннюю структуру списка. В списке используется архитектура “модель–представление–контроллер”. Это позволяет отделить данные (т.е. набор объектов) от их визуального представления (т.е. отображение столбцов с пунктами списка).

Класс JList отвечает за визуальное представление данных, и ему ничего не известно о способе их хранения. Он лишь извлекает информацию с помощью объекта, реализующего интерфейс ListModel.

```
public interface ListModel
{
   int getSize();
   Object getElementAt(int i);
   void addListDataListener(ListDataListener l);
   void removeListDataListener(ListDataListener l);
}
```

С помощью этого интерфейса компонент JList может получать значения счетчика пунктов и извлекать требуемый элемент. Кроме того, объект JList может выступать в качестве ListDataListener. В таком случае он получает сообщения об изменении состава списка и способен обновлять визуальное представление самого себя.

Зачем используется такая универсальная структура? Почему бы не хранить в составе JList вектор других объектов?

Обратите внимание на то, что в интерфейсе не указывается способ хранения данных. Более того, он даже не требует сохранять данные! Метод getElementAt() можно переопределить так, чтобы он вычислял требуемое значение (это показано в следующем примере). Такой подход полезен для представления очень большого набора данных без сохранения значений.

На рис. 6.3 показан еще один упрощенный вариант списка с перечнем *всех возможных трехбуквенных слов*.

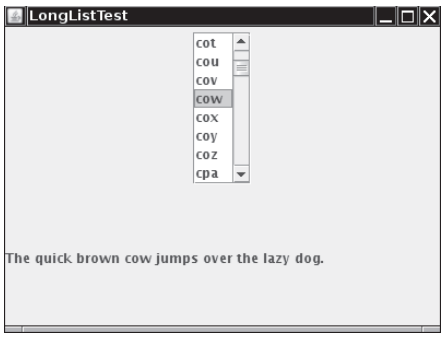


Рис. 6.3. Пример выбора значения из длинного списка

Всего существует 26×26×26=17 576 трехбуквенных сочетаний. Вместо хранения всех этих значений было бы разумнее вычислять их во время прокрутки списка.

Оказывается, реализовать такой способ довольно просто. Рутинные операции вставки и удаления обработчиков уже подготовлены в классе AbstractListModel, который является суперклассом используемого нами класса WordListModel. Остается только создать код методов getSize() и getElementAt().

```
class WordListModel extends AbstractListModel
{
   public WordListModel(int n) { length = n; }
   public int getSize() { return (int)Math.pow(26, length); }
   public Object getElementAt(int n)
   {
      // вычисление n-й строки
      ...
   }
   ...
}
```

Подробности вычисления *n*-й строки можно найти в листинге 6.2.

Теперь, имея модель, можно создать список с возможностью прокрутки вычисляемых элементов:

```
JList wordList = new JList(new WordListModel(3));
wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
JScrollPane scrollPane = new JScrollPane(wordList);
```

Основная идея заключается в том, что строки *не сохраняются*, а вычисляются. Причем генерируются только те строки, которые нужны в данный момент пользователю.

Необходимо выполнить еще одну установку. Мы должны сообщить элементу, что все пункты имеют фиксированную высоту и ширину. Проще всего установить размеры ячейки, определив *прототип значения ячейки* (prototype cell value):

```
wordList.setPrototypeCellValue("www");
```

Прототип используется для вычисления размеров всех ячеек. (Мы используем здесь строку `"www"` по той причине, что в большинстве шрифтов буква `"w"` является самой широкой из букв нижнего регистра.) Можно также задать размеры явным образом:

```
WordList.setFixedCellWidth(50);
WordList.setFixedCellHeight(15);
```

Если не задан прототип и не установлены фиксированные размеры, высота и ширина будут вычисляться отдельно для каждого элемента, а на это потребуется довольно много времени.

На практике длинные списки используются крайне редко, потому что пользователям очень сложно искать и выбирать в них требуемые элементы. Кроме того, набор выбранных пользователем пунктов имеет достаточно малый объем и может быть сохранен непосредственно в компоненте `JList`. Такой порядок работы позволяет программистам не рассматривать модель списка отдельно от самого компонента. С другой стороны, компонент `JList` оказывается совместимым с классами `JTable` и `JTree`.

Листинг 6.2. Содержимое файла `LongListTest.java`

```
1.  import java.awt.*;
2.
3.  import javax.swing.*;
4.  import javax.swing.event.*;
5.
6.  /**
7.   * Пример списка с динамической генерацией пунктов.
8.   * @version 1.23 2007-08-01
9.   * @author Cay Horstmann
10.  */
11. public class LongListTest
12. {
13.    public static void main(String[] args)
14.    {
15.         EventQueue.invokeLater(new Runnable()
16.              {
17.                   public void run()
18.                   {
19.                        JFrame frame = new LongListFrame();
20.                        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.                        frame.setVisible(true);
```

```
22.                   }
23.                });
24.    }
25. }
26.
27. /**
28.  * Этот фрейм содержит длинный список слов и текстовую
29.  * метку, содержащую предложение, в которое
30.  * включены выбранные слова.
31.  */
32. class LongListFrame extends JFrame
33. {
34.    public LongListFrame()
35.    {
36.       setTitle("LongListTest");
37.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
38.
39.       wordList = new JList(new WordListModel(3));
40.       wordList.setSelectionMode
41.          (ListSelectionModel.SINGLE_SELECTION);
42.       wordList.setPrototypeCellValue("www");
43.       JScrollPane scrollPane = new JScrollPane(wordList);
44.
45.       JPanel p = new JPanel();
46.       p.add(scrollPane);
47.       wordList.addListSelectionListener(new ListSelectionListener()
48.          {
49.             public void valueChanged(ListSelectionEvent evt)
50.             {
51.                StringBuilder word =
52.                   (StringBuilder) wordList.getSelectedValue();
53.                setSubject(word.toString());
54.             }
55.
56.          });
57.
58.       Container contentPane = getContentPane();
59.       contentPane.add(p, BorderLayout.NORTH);
60.       label = new JLabel(prefix + suffix);
61.       contentPane.add(label, BorderLayout.CENTER);
62.       setSubject("fox");
63.    }
64.
65.    /**
66.      *  Включение слова в состав предложения.
67.      *  @param word Новое подлежащее, включаемое в предложение
68.      */
69.    public void setSubject(String word)
70.    {
71.       StringBuilder text = new StringBuilder(prefix);
72.       text.append(word);
73.       text.append(suffix);
74.       label.setText(text.toString());
75.    }
76.
77.    private static final int DEFAULT_WIDTH = 400;
78.    private static final int DEFAULT_HEIGHT = 300;
```

```
79.    private JList wordList;
80.    private JLabel label;
81.    private String prefix = "The quick brown ";
82.    private String suffix = " jumps over the lazy dog.";
83. }
84.
85. /**
86.  * Модель, динамически генерирующая слова из n букв.
87.  */
88. class WordListModel extends AbstractListModel
89. {
90.    /**
91.      * Конструктор модели.
92.      * @param n Длина слова
93.      */
94.    public WordListModel(int n)
95.    {
96.         length = n;
97.    }
98.
99.    public int getSize()
100.   {
101.     return (int) Math.pow(LAST - FIRST + 1, length);
102.   }
103.
104.   public Object getElementAt(int n)
105.   {
106.     StringBuilder r = new StringBuilder();
107.      ;
108.     for (int i = 0; i < length; i++)
109.     {
110.       char c = (char)(FIRST + n % (LAST - FIRST + 1));
111.       r.insert(0, c);
112.       n = n / (LAST - FIRST + 1);
113.     }
114.     return r;
115.   }
116.
117.   private int length;
118.   public static final char FIRST = 'a';
119.   public static final char LAST = 'z';
120. }
```

API javax.swing.JList 1.2

- `JList(ListModel dataModel)`

  Создает список, в котором отображаются элементы согласно указанной модели.

- `Object getPrototypeCellValue()`
- `void setPrototypeCellValue(Object newValue)`

  Устанавливает или возвращает прототип значения ячейки, используемый для определения ширины и высоты каждой ячейки в списке. По умолчанию принимается значение `null`, в результате чего размеры вычисляются отдельно для каждого пункта списка.

- `void setFixedCellWidth(int width)`

  Если `width` больше 0, данный метод определяет ширину (в пикселях) всех ячеек в списке. По умолчанию `width` имеет значение `-1`, которое означает вычисление ширины каждой ячейки.

- `void setFixedCellHeight(int height)`

  Если `height` больше 0, данный метод определяет высоту (в пикселях) всех ячеек в списке. По умолчанию `width` имеет значение `-1`, которое означает вычисление высоты каждой ячейки.

API javax.swing.ListModel 1.2

- `int getSize()`

  Возвращает количество элементов в модели.

- `Object getElementAt(int position)`

  Возвращает элемент модели, соответствующий указанной позиции.

Вставка и удаление значений

Нельзя непосредственно редактировать набор значений списка. Нужно сначала обеспечить доступ к *модели* и только после этого выполнять операции удаления и добавления элементов. Следует отметить, что это не такая простая задача, как может показаться на первый взгляд. Итак, сначала нужно получить ссылку на модель списка:

```
ListModel model = list.getModel();
```

Пока это ничего не дает программисту, потому что, как было сказано ранее, интерфейс `ListModel` не содержит методов добавления и удаления элементов. Основная задача модели заключается в том, чтобы избежать необходимости *хранения* элементов.

Попробуем поступить иначе. Один из конструкторов списка получает в качестве параметра вектор объектов:

```
Vector<String> values = new Vector<String>();
values.addElement("quick");
values.addElement("brown");
. . .
JList list = new JList(values);
```

Теперь можно редактировать вектор, добавлять и удалять его элементы. Но список ничего не будет знать о таких изменениях, поэтому он никак не будет реагировать на них. В частности, список не будет обновляться при добавлении новых элементов. Следовательно, такой конструктор не поможет решить нашу задачу.

Вместо этого нужно создать объект `DefaultListModel`, заполнить его необходимыми значениями и связать со списком. Класс `DefaultListModel` реализует интерфейс `ListModel` и управляет коллекцией объектов.

```
DefaultListModel model = new DefaultListModel();
model.addElement("quick");
model.addElement("brown");
...
JList list = new JList(model);
```

Теперь можно добавлять значения в объект `model` и удалять их, объект будет уведомлять список о происходящих изменениях, а список, в свою очередь, изменять отображаемые данные:

```
model.removeElement("quick");
model.addElement("slow");
```

Как видите, имена методов в классе `DeafultListModel` отличаются от методов, применяемых в наборах данных.

Для внутреннего хранения значений в модели типа `DeafultListModel` используется вектор.



Некоторые конструкторы `JList` создают список из массива либо вектора объектов или строк. Напрашивается вывод, что эти конструкторы используют класс `DeafultListModel` для хранения значений пунктов. Однако это нет так. Они создают обычную модель, в которой отсутствует уведомление об изменении содержимого. Ниже приводится код конструктора для создания списка `JList` на основе класса `Vector`.

```
public JList(final Vector<?> listData)
{
    this (new AbstractListModel()
    {
     public int getSize() { return listData.size(); }
     public Object getElementAt(int i) { return listData.elementAt(i); }
    });
}
```

Это значит, что, если после создания списка вектор изменится, список будет содержать смесь старых и новых значений до тех пор, пока не будет полностью перерисован. (Ключевое слово `final` в приведенном выше конструкторе не накладывает запрет на изменение элементов списка, а только запрещает изменять значение ссылки `listData`. Это ключевое слово необходимо, потому что объект `listData` используется во внутреннем классе.)



javax.swing.JList 1.2

- `ListModel getModel()`

  Возвращает модель для данного списка.



javax.swing.DefaultListModel 1.2

- `void addElement(Object obj)`

  Добавляет объект в конце модели.

- `void removeElement(Object obj)`

  Удаляет первый экземпляр объекта из модели и возвращает значение `true`, если такой объект был найден, либо значение `false` в противном случае.

Отображение значений

Все приведенные до сих пор списки содержали строки. Но можно также создать список пиктограмм, передавая массив или вектор объектов типа `Icon`. Более того, можно создать список любых объектов и представить их на экране любым нужным вам способом.

Если строки и пиктограммы список `JList` отображает автоматически, то для вывода произвольных объектов необходимо включить в состав `JList` *объект отображения ячеек списка* (list cell renderer). В его роли может выступать экземпляр любого класса, реализующего интерфейс `ListCellRenderer`.

```
interface ListCellRenderer
{
  Component getListCellRendererComponent(JList list,
    Object value, int index,
    boolean isSelected, boolean cellHasFocus);
}
```

Метод, объявленный в данном интерфейсе, вызывается для каждой ячейки. Он возвращает компонент, предназначенный для отображения содержимого. Компонент помещается в позицию, соответствующую воспроизводимой ячейке.

Для реализации объекта отображения ячеек проще всего создать класс, расширяющий `JComponent`. Пример такого класса приведен ниже.

```
class MyCellRenderer extends JComponent implements ListCellRenderer
{
  public Component getListCellRendererComponent(JList list,
    Object value, int index, boolean isSelected,
    boolean cellHasFocus)
  {
    // Хранение информации, необходимой для
    // визуализации и управления размерами.
    return this;
  }
  public void paintComponent(Graphics g)
  {
    // Код, предназначенный для визуализации.
  }
  public Dimension getPreferredSize()
  {
    // Код, предназначенный для изменения размеров.
  }
  // Переменные экземпляра.
}
```

В листинге 6.3 приведен пример программы выбора шрифта. Имя в списке отображается соответствующим шрифтом (рис. 6.4). Для вывода названия каждого шрифта используется метод `paintComponent()`. При этом необходимо учитывать цвет. Требуемую информацию мы получаем с помощью методов `getForeground()`/`getBackground()` и `getSelectionForeground()`/`getSelectionBackground()` класса `JList`. В методе `getPrefferedSize()` вычисляется размер строки. Способ получения размера подробно описывается в главе 7 первого тома.



Рис. 6.4. Пример списка с отображением разных шрифтов

Для установки объекта отображения ячеек нужно вызвать метод `setCellRenderer()`:

```
fontList.setCellRenrderer(new FontCellRenderer);
```

Теперь все элементы списка будут выводиться с помощью объекта отображения ячеек.

Существует более простой способ написания пользовательских объектов отображения ячеек. Если изображение, подлежащее выводу, содержит текст, пиктограмму и если при этом нужно учитывать цвет, то все необходимые эффекты можно получить с помощью класса `JLabel`. Например, для отображения названия шрифта этим же шрифтом подходит описанный ниже объект отображения.

```
class FontCellRenderer extends JLabel implements ListCellRenderer
{
  public Component getListCellRendererComponent(JList list,
    Object value, int index, boolean isSelected,
    boolean cellHasFocus)
  {
    JLabel label = new JLabel();
    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;
  }
}
```

Обратите внимание, что нам не пришлось создавать собственные методы `paintComponent()` или `getPrefferedSize()`, поскольку класс `JLabel` уже содержит их. Программисту остается только задать параметры метки, т.е. указать текст, шрифт и цвет.

Этот код удобно использовать в тех случаях, когда какой-то существующий компонент (в данном примере `JLabel`) поддерживает все функции, необходимые для визуализации пунктов списка.

Мы могли бы использовать компонент `JLabel` в нашей программе, однако мы специально предложили более обобщенный код, чтобы вы могли изменить его, если понадобится нарисовать что-нибудь в пунктах списка.



Создавать новый компонент при каждом вызове метода `getListCellRendererComponent()` вряд ли оправдано. Если пользователю приходится прокручивать список, содержащий большое количество пунктов, каждый раз будет создаваться новый компонент. Более эффективное и надежное решение состоит в переконфигурировании существующего компонента.

Листинг 6.3. Содержимое файла `ListRenderingTest.java`

```
1.  import java.util.*;
2.  import java.awt.*;
3.
4.  import javax.swing.*;
5.  import javax.swing.event.*;
```

```
6.  /**
7.   *  Пример использования объекта отображения ячеек списка.
8.   * @version 1.23 2007-08-01
9.   * @author Cay Horstmann
10.  */
11. public class ListRenderingTest
12. {
13.    public static void main(String[] args)
14.    {
15.         EventQueue.invokeLater(new Runnable()
16.              {
17.                  public void run()
18.                   {
19.                       JFrame frame = new LongListFrame();
20.                       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.                       frame.setVisible(true);
22.                   }
23.               });
24.    }
25. }
26.
27. /**
28.  * Этот фрейм содержит список шрифтов и текстовую область,
29.  * в которой текст отображается выбранным шрифтом.
30.  */
31. class ListRenderingFrame extends JFrame
32. {
33.    public ListRenderingFrame()
34.    {
35.       setTitle("ListRenderingTest");
36.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
37.
38.       ArrayList<Font> fonts = new ArrayList<Font>();
39.       final int SIZE = 24;
40.       fonts.add(new Font("Serif", Font.PLAIN, SIZE));
41.       fonts.add(new Font("SansSerif", Font.PLAIN, SIZE));
42.       fonts.add(new Font("Monospaced", Font.PLAIN, SIZE));
43.       fonts.add(new Font("Dialog", Font.PLAIN, SIZE));
44.       fonts.add(new Font("DialogInput", Font.PLAIN, SIZE));
45.       fontList = new JList(fonts.toArray());
46.       fontList.setVisibleRowCount(4);
47.       fontList.setSelectionMode
48.          (ListSelectionModel.SINGLE_SELECTION);
49.       fontList.setCellRenderer(new FontCellRenderer());
50.       JScrollPane scrollPane = new JScrollPane(fontList);
51.
52.       JPanel p = new JPanel();
53.       p.add(scrollPane);
54.       fontList.addListSelectionListener(new ListSelectionListener()
55.          {
56.             public void valueChanged(ListSelectionEvent evt)
57.             {
58.                Font font = (Font) fontList.getSelectedValue();
59.                text.setFont(font);
60.             }
61.
62.          });
```

```
63.       Container contentPane = getContentPane();
64.       contentPane.add(p, BorderLayout.SOUTH);
65.       text = new
66.         JTextArea("The quick brown fox jumps over the lazy dog");
67.       text.setFont((Font) fonts.get(0));
68.       text.setLineWrap(true);
69.       text.setWrapStyleWord(true);
70.       contentPane.add(text, BorderLayout.CENTER);
71.    }
72.
73.    private JTextArea text;
74.    private JList fontList;
75.    private static final int DEFAULT_WIDTH = 400;
76.    private static final int DEFAULT_HEIGHT = 300;
77. }
78.
79. /**
80.  * Класс отображения ячеек для объектов Font.
81.  * Он позволяет выводить название шрифта этим же шрифтом.
82.  */
83. class FontCellRenderer extends JPanel implements ListCellRenderer
84. {
85.    public Component getListCellRendererComponent(JList list,
86.       Object value, int index, boolean isSelected,
87.       boolean cellHasFocus)
88.    {
89.       font = (Font) value;
90.       background = isSelected ?
91.         list.getSelectionBackground() : list.getBackground();
92.       foreground = isSelected ?
93.         list.getSelectionForeground() : list.getForeground();
94.       return this;
95.    }
96.
97.    public void paintComponent(Graphics g)
98.    {
99.       String text = font.getFamily();
100.      FontMetrics fm = g.getFontMetrics(font);
101.      g.setColor(background);
102.      g.fillRect(0, 0, getWidth(), getHeight());
103.      g.setColor(foreground);
104.      g.setFont(font);
105.      g.drawString(text, 0, fm.getAscent());
106.   }
107.
108.   public Dimension getPreferredSize()
109.   {
110.      String text = font.getFamily();
111.      Graphics g = getGraphics();
112.      FontMetrics fm = g.getFontMetrics(font);
113.      return new Dimension(fm.stringWidth(text), fm.getHeight());
114.   }
115.
116.   private Font font;
117.   private Color background;
118.   private Color foreground;
119. }
```

API javax.swing.JList 1.2

- `Color getBackground()`
  Возвращает цвет фона для невыбранных пунктов списка.
- `Color getSelectionBackground()`
  Возвращает цвет фона для выбранных пунктов списка.
- `Color getForeground()`
  Возвращает цвет переднего плана для невыбранных пунктов списка.
- `Color getSelectionForeground()`
  Возвращает цвет переднего плана для выбранных пунктов списка.
- `void setCellRenderer(ListCellRenderer cellRenderer)`
  Устанавливает объект отображения ячеек, который используется для вывода пунктов списка.

API javax.swing.ListCellRenderer 1.2

- `Component getListCellRendererComponent(JList list, Object item, int index, boolean isSelected, boolean hasFocus)`
  Возвращает компонент, метод `paint()` которого будет использоваться для визуализации пунктов списка. Если размер ячеек не фиксирован, то этот компонент должен также реализовать метод `getPrefferedSize()`.

Параметры:		
	`list`	Список, ячейки которого должны быть отображены
	`item`	Отображаемый пункт списка
	`index`	Индекс, определяющий расположение пункта в составе модели
	`isSelected`	Значение `true`, если указанный пункт был выбран
	`hasFocus`	Значение `true`, если указанный пункт обладает фокусом ввода

Таблицы

Компонент `JTable` используется для отображения таблицы, т.е. двухмерной сетки объектов. Таблицы широко применяются для создания пользовательских интерфейсов, поэтому команда разработчиков библиотеки Swing уделила большое внимание созданию соответствующего элемента управления. 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. Однако для формирования специальных таблиц придется написать дополнительный код и должным образом задать их внешний вид и поведение.

В этом разделе рассматриваются методы создания простых таблиц, приемы работы с ними и наиболее распространенные способы настройки. Как и при работе с другими более сложными компонентами Swing, в рамках одного раздела невозможно подробно описать все эти методы. Для углубленного изучения этой темы рекомендуется использовать книгу Дэвида М. Гери (David M. Geary) *Graphic Java 2: Mastering the JFC, Volume II: Swing, 3rd Edition*, (Prentice Hall PTR, 1999 г.) или книгу Кима Топли (Kim Topley) *Core Java Foundation Classes* (Prentice Hall, 1998 г.).

Простая таблица

Подобно `JList`, компонент `JTable` не хранит свои данные, а получает их от *модели таблицы*. Класс `JTable` содержит конструктор, который включает двухмерный массив объектов в используемую по умолчанию модель. Именно эта стратегия применяется в нашем первом примере. Далее мы рассмотрим модели таблиц более подробно.

На рис. 6.5 показана типичная таблица с описаниями свойств планет Солнечной системы. Для справки: столбец `Gaseous` содержит значение `true`, если планета состоит в основном из водорода и гелия, или значение `false` в противном случае. Значения в столбце `Color` пока не имеют какого-то смысла, этот столбец понадобится нам в следующем примере.

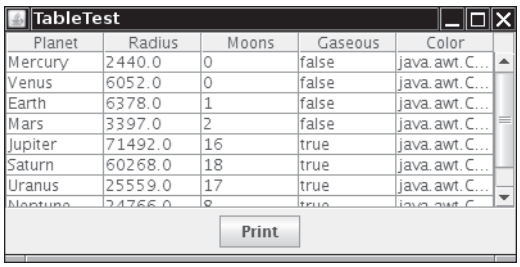


Рис. 6.5. Пример простой таблицы

Как показано в листинге 6.4, 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 },
   . . .
}
```



Здесь мы пользуемся преимуществом автоматического преобразования простых типов. Значения во втором, третьем и четвертом столбцах преобразуются в объекты `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=...]`.

Имена столбцов задаются в отдельном массиве строк:

```
String[] columnNames = {  "Planet", "Radius", "Moons", "Gaseous", "Color" };
```

Затем на основании массива значений типа `Object` и массива с именами столбцов создается таблица. После этого за счет вставки таблицы в панель `JScrollPane` стандартным образом формируются полосы прокрутки:

```
JTable table = new JTable(cells, columnNames);
JScrollPane pane = new JScrollPane(table);
```

К удивлению, столь просто полученная таблица имеет очень привлекательный внешний вид! При достаточно большом уменьшении таблицы автоматически появляется полоса прокрутки. Обратите внимание, что при прокрутке таблицы вниз имена столбцов всегда видны.

Попробуйте щелкнуть кнопкой мыши на одном из столбцов и перетащить его в другое место. Обратите внимание на то, что столбец можно визуально отделить от других столбцов небольшим зазором (рис. 6.6). Такое переупорядочение столбцов возможно *только в представлении таблицы* и никак не влияет на модель данных.

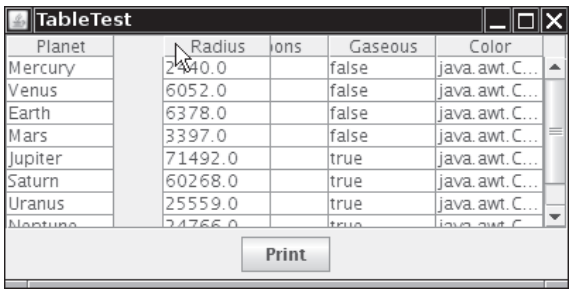


Рис. 6.6. Перемещение столбца

Для *изменения размеров столбца* поместите указатель мыши на границе раздела между двумя столбцами. После того как указатель мыши примет форму двойной стрелки, перетащите границу раздела столбцов в нужное место (рис. 6.7).

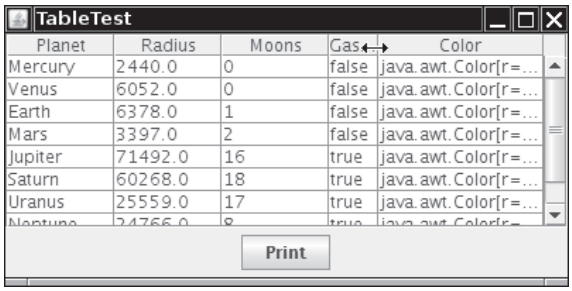


Рис. 6.7. Изменение размеров столбца

Выбор строки осуществляется щелчком мыши в любом месте строки. При этом выделенные строки обозначаются изменением цвета. Более подробно обработка событий выбора описывается далее в данном разделе. В рассматриваемом примере допускается редактирование полей, но эти изменения не сохраняются. Создавая свою программу, следует либо вообще запретить редактирование, либо организовать обработку событий редактирования и обновления модели данных. Эти вопросы подробно рассматриваются несколько позже.

И, наконец, щелкните на заголовке столбца, чтобы автоматически отсортировать строки. Щелкните еще раз, и сортировка будет произведена в обратном порядке. Это поведение управляется посредством вызова метода

```
table.setAutoCreateRowSorter(true);
```

Таблицу можно вывести на печать, используя для этого метод `print()`:

```
table.print();
```

В ответ отобразится диалоговое окно, и таблица будет передана на принтер. Варианты вывода на печать рассматриваются в главе 7.



Если изменить фрейм `TableTest` так, чтобы по высоте он был выше таблицы, вы увидите серую область под таблицей. В отличие от компонентов `JList` и `JTree`, таблица не заполняет область просмотра панели прокрутки. Поэтому, если нужно обеспечить поддержку функции перетаскивания, могут возникнуть проблемы. (Более подробно об этом речь пойдет в главе 7.) В нашем случае, вызовите следующий метод:

```
table.setFillsViewportHeight(true);
```

Листинг 6.4. Содержимое файла `PlanetTable.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import javax.swing.*;
4.
5.  /**
6.   * Программа, демонстрирующая отображение простой таблицы.
7.   * @version 1.11 2007-08-01
8.   * @author Cay Horstmann
9.   */
10.
11. public class PlanetTable
12. {
13.    public static void main(String[] args)
14.    {
15.         EventQueue.invokeLater(new Runnable()
16.              {
17.                   public void run()
18.                   {
19.                        JFrame frame = new LongListFrame();
20.                        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.                        frame.setVisible(true);
22.                   }
23.               });
24.    }
25. }
26.
27. /**
28.  *  Данный фрейм содержит таблицу с данными о планетах.
29.  */
30. class PlanetTableFrame extends JFrame
31. {
32.    public PlanetTableFrame()
33.    {
34.       setTitle("PlanetTable");
35.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
36.       final JTable table = new JTable(cells, columnNames);
37.       table.setAutoCreateRowSorter(true);
38.       add(new JScrollPane(table), BorderLayout.CENTER);
39.       JButton printButton = new JButton("Print");
40.       printButton.addActionListener(new ActionListener()
41.          {
42.             public void actionPerformed(ActionEvent event)
43.             {
44.                try
45.                {
46.                   table.print();
47.                }
```

```
48.              catch (java.awt.print.PrinterException e)
49.              {
50.                 e.printStackTrace();
51.              }
52.           }
53.        });
54.       JPanel buttonPanel = new JPanel();
55.       buttonPanel.add(printButton);
56.       add(buttonPanel, BorderLayout.SOUTH);
57.    }
58.
59.    private Object[][] cells = { { "Mercury", 2440.0,  0, false, Color.yellow },
60.         { "Venus", 6052.0, 0, false, Color.yellow },
61.         { "Earth", 6378.0, 1, false, Color.blue },
62.         { "Mars", 3397.0, 2, false, Color.red },
63.         { "Jupiter", 71492.0, 16, true, Color.orange },
64.         { "Saturn", 60268.0, 18, true, Color.orange },
65.         { "Uranus", 25559.0, 17, true, Color.blue },
66.         { "Neptune", 24766.0, 8, true, Color.blue },
67.         { "Pluto", 1137.0, 1, false, Color.black } };
68.
69.    private String[] columnNames = { "Planet", "Radius", "Moons", "Gaseous", "Color" };
70.
71.    private static final int DEFAULT_WIDTH = 400;
72.    private static final int DEFAULT_HEIGHT = 200;
73. }
```

API javax.swing.JTable 1.2

- `JTable(Object[][] entries, Object[] columnNames)`

  Создает таблицу с используемой по умолчанию моделью.

- `void print()` 5.0

  Отображает диалоговое окно, предназначенное для работы с принтером, и направляет таблицу на печать.

- `boolean getAutoCreateRowSorter()` 6.0
- `void setAutoCreateRowSorter(boolean newValue)` 6.0

  Возвращает или устанавливает свойство `autoCreateRowSorter`. По умолчанию оно равно `false`. Если установлено именно это значение, то при любом изменении модели будет автоматически устанавливаться сортировщик строк по умолчанию.

- `boolean getFillsViewportHeight()` 6.0
- `void setFillsViewportHeight(boolean newValue)` 6.0

  Возвращает или устанавливает свойство `fillsViewportHeight`. По умолчанию оно равно `false`. Если установлено именно это значение, то таблица всегда будет заполнять включающую область просмотра.

Модели таблиц

Описанные в предыдущем разделе данные таблицы хранились в виде двухмерного массива. Однако в общем случае рекомендуется использовать собственную модель таблицы.

Модель таблицы создать довольно просто, поскольку для этого предусмотрен специальный класс `AbstractTableModel` с реализацией большинства необходимых методов. Программисту остается создать лишь три метода:

```
public int getRowCount();
public int getColumnCount();
public Object getValueAt(int row, int column);
```

Метод `getValueAt()` можно реализовать несколькими способами. Например, если необходимо отобразить содержимое `RowSet`, содержащего 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;
      }
}
```

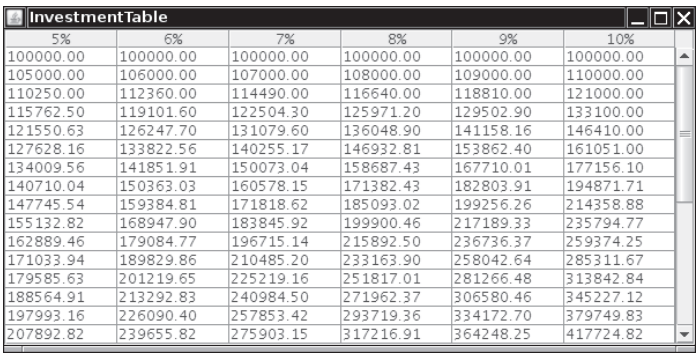
Наша же программа еще проще. Мы создаем таблицу с отображением вычисленных значений, а именно роста инвестиций для разных учетных ставок (рис. 6.8).

В этом примере метод `getValueAt()` вычисляет соответствующее значение и форматирует его показанным ниже способом.

```
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);
}
```

Методы `getRowCount()` и `getColumnCount()` просто возвращают количество строк и столбцов соответственно:

```
public int getRowCount() { return years; }
public int getColumnCount() { return maxRate - minRate + 1; }
```



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, отражающей рост инвестиций для разных учетных ставок

Если заголовки столбцов не заданы явно, то метод `getColumnName()` класса `AbstractTableModel` по умолчанию использует для них имена `A`, `B`, `C` и т.д. Для изменения этих имен нужно переопределить метод `getColumnName()`. В этом примере в качестве заголовка столбца используется учетная ставка:

```
public String getColumnName(int c) { return (c + minRate) + "%"; }
```

Полный код программы приведен в листинге 6.5.

Листинг 6.5. Содержимое файла `InvestmentTable.java`

```
1.  import java.awt.*;
2.
3.  import javax.swing.*;
4.  import javax.swing.table.*;
5.
6.  /**
7.   * Пример создания таблицы на основе модели.
8.   * @version 1.02 2007-08-01
9.   * @author Cay Horstmann
10.  */
11.
12. public class InvestmentTable
13. {
14.    public static void main(String[] args)
15.    {
16.         EventQueue.invokeLater(new Runnable()
17.              {
18.                   public void run()
19.                   {
20.                        JFrame frame = new LongListFrame();
21.                        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22.                        frame.setVisible(true);
23.                   }
24.               });
25.    }
26. }
27.
28. /**
29.  * Фрейм, содержащий таблицу с данными об инвестициях.
30.  */
31. class InvestmentTableFrame extends JFrame
32. {
33.    public InvestmentTableFrame()
34.    {
35.       setTitle("InvestmentTable");
36.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
37.
38.       TableModel model = new InvestmentTableModel(30, 5, 10);
39.       JTable table = new JTable(model);
40.       add(new JScrollPane(table));
41.    }
42.
43.    private static final int DEFAULT_WIDTH = 600;
44.    private static final int DEFAULT_HEIGHT = 300;
45. }
46.
47. /**
```

```
48.   * Модель таблицы вычисляет данные для отображения в ячейках
49.   * для каждого запроса. В таблице показан рост инвестиций
50.   * за указанное количество лет для разных учетных ставок
51.   */
52.  class InvestmentTableModel extends AbstractTableModel
53.  {
54.     /**
55.      * Создание модели таблицы инвестиций.
56.      * @param y Количество лет
57.      * @param r1 Наименьшая учетная ставка
58.      * @param r2 Наибольшая учетная ставка
59.      */
60.     public InvestmentTableModel(int y, int r1, int r2)
61.     {
62.        years = y;
63.        minRate = r1;
64.        maxRate = r2;
65.     }
66.     public int getRowCount()
67.    {
68.          return years;
69.    }
70.     public int getColumnCount()
71.    {
72.          return maxRate - minRate + 1;
73.    }
74.     public Object getValueAt(int r, int c)
75.     {
76.        double rate = (c + minRate) / 100.0;
77.        int nperiods = r;
78.        double futureBalance = INITIAL_BALANCE * Math.pow(1 + rate, nperiods);
79.        return String.format("%.2f", futureBalance);
80.     }
81.     public String getColumnName(int c)
82.     {
83.          return (c + minRate) + "%";
84.     }
85.
86.     private int years;
87.     private int minRate;
88.     private int maxRate;
89.
90.     private static double INITIAL_BALANCE = 100000.0;
91.  }
```

API javax.swing.table.TableModel 1.2

- `int getRowCount()`
- `int getColumnCount()`

  Возвращает количество строк и столбцов в модели таблицы.

- `Object getValueAt(int row, int column)`

  Возвращает значение в данной строке и столбце.

- `void setValueAt(Object newValue, int row, int column)`

  Устанавливает новое значение в данной строке и столбце.

- `boolean isCellEditable(int row, int column)`
  Возвращает `true`, если ячейка в данной строке и столбце доступна для редактирования.
- `String getColumnName(int column)`
  Возвращает заголовок столбца.

Работа со строками и столбцами

В этом разделе рассматриваются способы работы со строками и столбцами таблицы. Таблица, сформированная средствами Swing, имеет несимметричную структуру, т.е. для строк и столбцов допускается выполнение разных операций. Причина в том, что компонент библиотеки Swing, реализующий таблицу, предназначен для отображения строк одинаковой структуры, например, информации из таблиц базы данных. Эта асимметрия демонстрируется на примерах данного раздела.

Классы столбцов

В следующем примере снова рассматривается таблица с данными о планетах, но теперь особое внимание уделяется типам данных в столбцах. Указанный ниже метод модели таблицы возвращает класс с описанием типа столбца:

```
Class<?> getColumnClass(int columnIndex)
```

В табл. 6.1 приведены классы отображения, которые используются по умолчанию в классе `JTable`.

Флажки опций и изображения показаны на рис. 6.9. Авторы выражают свою благодарность Джиму Эвинсу (Jim Evins) за любезно представленные рисунки планет (`http://www.shaught.com/JimsCoolIcons/Planets`).

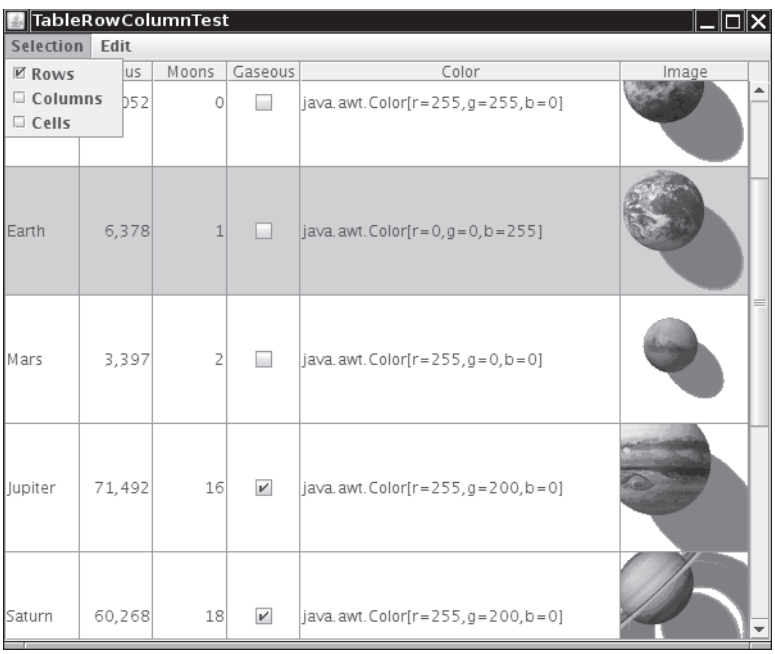


Рис. 6.9. Пример таблицы с объектами отображения ячеек

Таблица 6.1. Действия по умолчанию при визуализации

Тип	Визуализируется как
`Boolean`	Флажок опции
`Icon`	Изображение
`Object`	Строка

Чтобы визуализировать другие типы, можно установить специальный класс отображения (см. раздел "Визуализация и редактирование ячеек" далее в главе).

Доступ к столбцам таблицы

Класс `JTable` хранит информацию обо всех столбцах таблицы в объектах типа `TableColumn`. Объект `TableColumnModel` управляет столбцами. (На рис. 6.10 показаны отношения между наиболее важными табличными классами.) Если столбцы таблицы не предполагается динамически перемещать или вставлять, то в таком случае не придется очень интенсивно использовать класс модели столбцов таблицы.

Наиболее распространенным варрантом использования модели столбцов является получение объекта `TableColumn`:

```
int columnIndex = . . .:
TableColumn column = table.getColumnModel().getColumn(columnIndex);
```

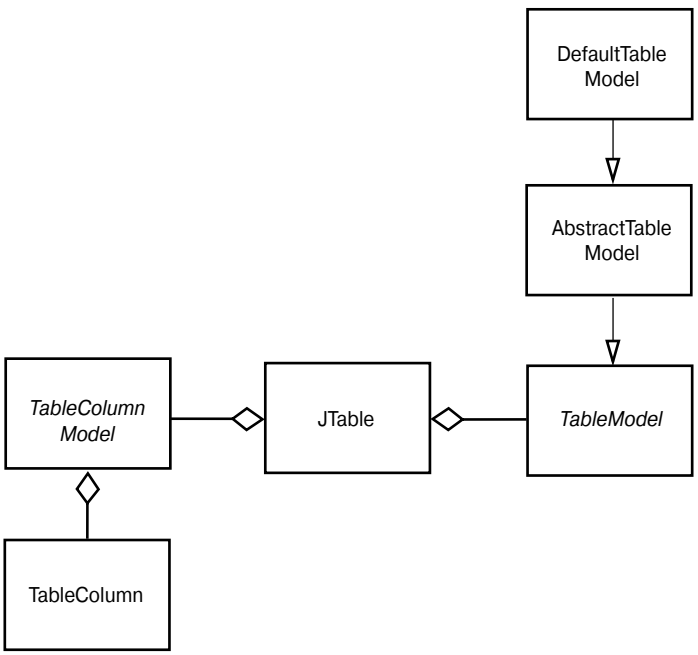


Рис. 6.10. Отношения между табличными классами

Изменение размеров столбцов

Класс `TableColumn` позволяет управлять размерами столбцов. 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 таблицы для размещения столбцов.

Метод `void setResizable(boolean resizable)` указывает, допустимо ли изменение размеров столбца.

Изменить размер столбца из программы можно, вызвав метод `void setWidth(int width)`.

При изменении размера одного столбца общий размер таблицы по умолчанию остается прежним. Конечно, при этом увеличиваются или уменьшаются размеры остальных столбцов. По умолчанию такие изменения происходят для всех столбцов, которые находятся справа от данного. Благодаря такому правилу программист может указать нужные размеры для всех столбцов слева направо.

В табл. 6.2 перечислены варианты поведения таблицы при изменении размеров столбца. Их можно указать с помощью метода `void setAutoResizeMode(int mode)` класса `JTable`.

Таблица 6.2. Режимы изменения размеров

Режим	Поведение
`AUTO_RESIZE_OFF`	Размеры остальных столбцов остаются прежними; изменяется размер таблицы.
`AUTO_RESIZE_NEXT_COLUMN`	Изменяется только размер следующего столбца.
`AUTO_RESIZE_SUBSEQUENT_COLUMNS`	Происходит равномерное изменение размеров для всех столбцов, находящихся справа (используется по умолчанию).
`AUTO_RESIZE_LAST_COLUMN`	Изменяется размер только последнего столбца.
`AUTO_RESIZE_ALL_COLUMNS`	Размеры всех столбцов таблицы изменяются равномерно (использовать данный режим не рекомендуется, так как при этом усложняется настройка размеров для нескольких столбцов).

Изменение размеров строк

Высота строк управляется непосредственно классом `JTable`. Если высота ячеек выше заданной по умолчанию, то ее придется указать явно:

```
table.setRowHeight(height);
```

По умолчанию все строки таблицы имеют одинаковую высоту, но данную установку можно изменить с помощью метода `setRowHeight()`:

```
table.setRowHeight(row, height);
```

Фактическая высота строки задается перечисленными выше методами минус размер границы. По умолчанию этот размер равен 1, но можно указать любое другое значение:

```
table.setRowMargin(margin);
```

Выбор строк, столбцов и ячеек

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 показано на рис. 6.9. Для отмены режима выбора строк следует использовать метод

```
table.setRowSelectionAllowed(false)
```

В режиме выбора строк можно разрешить пользователям выбирать только одну строку, несколько смежных строк или несколько несмежных строк. Для этого нужно получить *модель выбора* и использовать ее метод `setSelectionMode()`:

```
table.getSelectionModel().setSelectionMode(mode);
```

В параметре `mode` можно указывать приведенные ниже значения:

```
ListSelectionModel.SINGLE_SELECTION
ListSelectionModel.SINGLE_INTERVAL_SELECTION
ListSelectionModel.MULTIPLE_INTERVAL_SELECTION
```

По умолчанию режим выбора строк отключен, и его можно включить с помощью приведенного ниже метода:

```
table.setColumnSelectionAllowed(true);
```

Одновременное включение режимов выбора строк и столбцов эквивалентно включению режима выбора ячеек (рис. 6.11). Этот режим можно указать явно, используя приведенный ниже метод:

```
table.setCellSelectionEnabled(true);
```

Запустите программу, код которой представлен в листинге 6.6, чтобы увидеть процесс выбора ячеек в действии. Разрешите выбор строки, столбца или ячейки в меню Selection (Выбор) и посмотрите, как изменяется поведение при выборе.

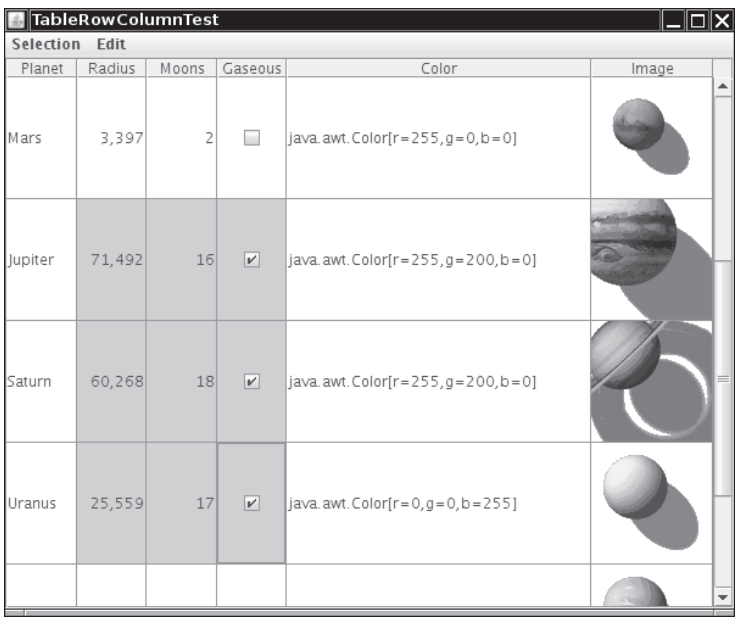


Рис. 6.11. Выбор ячеек

С помощью методов getSelectedRows и getSelectedColumns можно узнать, какие строки и столбцы были выбраны. Оба метода возвращают массив int[] индексов выбранных элементов. Обратите внимание на то, что значения индексов берутся из представления таблицы, а не из базовой модели таблицы. Попытайтесь выбрать строки и столбцы, затем перетащите столбцы в разные места и отсортируйте строки, щелкая на заголовках столбцов. С помощью элемента Print Selection (Выбор для печати) посмотрите, какие строки и столбцы отображаются как выбранные.

Если вам необходимо преобразовать значения индексов таблицы в значения индексов модели таблицы, используйте методы convertRowIndexToModel и convertColumnIndexToModel класса JTable.

Сортировка строк

Как было показано в нашем первом примере таблицы, можно без труда добавить функцию сортировки строк в JTable посредством вызова метода setAutoCreateRowSorter. Тем не менее, чтобы получить больше возможностей для управления сортировкой, необходимо установить и настроить объект TableRowSorter<M> в JTable. Параметр типа M указывает тип модели; это должен быть подтип интерфейса TableModel.

```
TableRowSorter<TableModel> sorter = new TableRowSorter<TableModel>(model);
table.setRowSorter(sorter);
```

Некоторые столбцы должны быть исключены из сортировки (например, столбец с изображением в наших данных о планетах). Отключить сортировку можно посредством вызова метода

```
sorter.setSortable(IMAGE_COLUMN, false);
```

Можно также установить специальный разделитель для каждого столбца. В нашем примере мы будем сортировать цвета в столбце Color, поскольку мы предпочитаем голубой и зеленый цвета красному. Если щелкнуть на столбце Column, можно увидеть, что голубые планеты окажутся внизу таблицы. Это достигается посредством следующего кода:

```
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 вы не определите разделитель для столбца, порядок сортировки будет определяться так, как описано ниже.

1. Если классом столбца является String, необходимо использовать подборщик, возвращенный методом Collator.getInstance(). Он сортирует строки таким способом, который подходит для текущей местной специфики. (Подробную информацию о местных спецификах и подборщиках можно найти в главе 5.)
2. Если класс столбца реализует интерфейс Comparable, используйте метод compareTo.
3. Если для компаратора был установлен TableStringConverter, сортировку строк, возвращенных методом toString преобразователя, необходимо выпол-

нять посредством подборщика по умолчанию. Если вы хотите применять этот подход, определите преобразователь следующим образом:

```
sorter.setStringConverter(new TableStringConverter()
        {
            public String toString(TableModel model, int row, int column)
            {
                Object value = model.getValueAt(row, column);
                 преобразуйте значение в строку и верните ее
            }
        });
```

4. В противном случае вызовите метод `toString` в значениях ячеек и отсортируйте их с помощью подборщика по умолчанию.

Фильтрация строк

Помимо сортировки строк, `TableRowSorter` может избирательно скрывать строки — этот процесс называется *фильтрацией* (filtering). Чтобы активизировать функцию фильтрации, необходимо установить `RowFilter`. Например, для включения всех строк, содержащих хотя бы один спутник, потребуется вызвать метод

```
sorter.setRowFilter(RowFilter.numberFilter(ComparisonType.NOT_EQUAL, 0, MOONS_COLUMN));
```

Здесь мы используем предварительно определенный фильтр — фильтр чисел. Чтобы создать такой фильтр, понадобится следующее.

- Тип сравнения (один для `EQUAL`, `NOT_EQUAL`, `AFTER` или `BEFORE`).
- Объект подкласса `Number` (например, `Integer` или `Double`). Допускаются только те объекты, которые имеют тот же класс, что и у данного объекта `Number`.
- Ни одного или несколько значений индекса столбца. Если значения не будут заданы, поиск будет производиться по всем столбцам.

Статический метод `RowFilter.dateFilter` создает фильтр данных подобным образом. Отличие заключается в том, что вы задаете объект `Date` вместо объекта `Number`.

Наконец, статический метод `RowFilter.regexFilter` создает фильтр, осуществляющий поиск строк, совпадающих с регулярным выражением. Например:

```
sorter.setRowFilter(RowFilter.regexFilter(".*[^s]$", PLANET_COLUMN));
```

отображает только те планеты, название которых не заканчивается на “`s`”. (Сведения о регулярных выражениях можно найти в главе 1.)

Вы можете также комбинировать фильтры посредством методов `andFilter`, `orFilter` и `notFilter`. Чтобы отфильтровать планеты, название которых не заканчивается на “`s`”, и которые имеют как минимум один спутник, можно использовать такую комбинацию фильтра:

```
sorter.setRowFilter(RowFilter.andFilter(Arrays.asList(
    RowFilter.regexFilter(".*[^s]$", PLANET_COLUMN),
    RowFilter.numberFilter(ComparisonType.NOT_EQUAL, 0, MOONS_COLUMN))));
```



Обратите внимание, что методы `andFilter` и `orFilter` не используют аргументы переменных, а один параметр типа `Iterable`.

Чтобы реализовать собственный фильтр, необходимо предоставить подкласс `RowFilter` и реализовать метод `include`, чтобы показать, какие строки необходимо

отобразить. Сделать это легко, однако из-за обобщенности класса RowFilter это становится несколько сложно.

Класс RowFilter<M, I> имеет два параметра: типы модели и идентификатор строк. При работе с таблицами модель всегда является подтипом TableModel, а идентификатор всегда имеет тип Integer. (Когда-нибудь остальные компоненты тоже будут поддерживать фильтрацию строк. Например, чтобы отфильтровать строки в JTree, может потребоваться класс RowFilter<TreeModel, TreePath>.)

Фильтр строк должен реализовывать следующий метод:

```
public boolean include(RowFilter.Entry<? extends M, ? extends I> entry)
```

Класс RowFilter предоставляет методы для получения модели, идентификатора строк и значения в данном индексе. Таким образом, фильтрацию можно производить как по идентификатору строк, так и по их содержимому.

Например, следующий фильтр отобразит каждую вторую строку:

```
RowFilter<TableModel, Integer> filter = new RowFilter<TableModel, Integer>()
    {
       public boolean include(Entry<? extends TableModel, ? extends Integer> entry)
       {
          return entry.getIdentifier() % 2 == 0;
       }
    };
```

Если вы хотите включить только те планеты, которые содержат четное количество спутников, вы можете попробовать использовать следующее:

```
((Integer) entry.getValue(MOONS_COLUMN)) % 2 == 0
```

В нашей программе мы позволяем пользователю скрывать произвольные строки. Индексы скрытых строк мы храним в наборе. Фильтр строк включает все строки, индекс которых не находится в данном наборе.

Механизм фильтрации не предназначался для фильтров, критерий которых изменяется с течением времени. В нашей программе мы повторяем вызов

```
sorter.setRowFilter(filter);
```

всякий раз, когда изменяется набор скрытых строк. При установке фильтра он применяется незамедлительно.

Сокрытие и отображение столбцов

Как было показано в предыдущем разделе, вы можете отфильтровывать строки таблицы или по их содержимому, или по их идентификатору. Чтобы скрыть столбцы, понадобится совершенно другой механизм.

Метод removeColumn() класса JTable позволяет удалить столбец (параметр TableColumn) из представления таблицы, т.е. скрыть его от пользователя, оставив в составе модели. Ниже показан способ извлечения фактического объекта, описывающего столбец таблицы, из модели по известному номеру (например, полученному с помощью метода getSelectedColumns()).

```
TableColumnModel columnModel = table.getColumnModel();
TableColumn column = columnModel.getColumn(i);
table.removeColumn(column);
```

Если этот объект запомнить, то впоследствии его можно вернуть обратно:

```
table.addColumn(column);
```

Этот метод добавляет столбец в конец таблицы, а для указания нужного расположения следует использовать метод `moveColumn()`.

Кроме того, создав объект `TableColumn`, можно сформировать новый столбец, который соответствует индексу в модели таблицы:

```
table.addColumn(new TableColumn(modelColumnIndex));
```

Таким образом, в таблице можно создать несколько столбцов, которые будут представлять один и тот же столбец из модели.

Программа, код которой представлен в листинге 6.6, показывает выбор и фильтрацию строк и столбцов.

Листинг 6.6. Содержимое файла `TableSelectionTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.util.*;
4.  import javax.swing.*;
5.  import javax.swing.table.*;
6.
7.  /**
8.   * Эта программа демонстрирует выбор, добавление и удаление строк и столбцов.
9.   * @version 1.03 2007-08-01
10.  * @author Cay Horstmann
11.  */
12. public class TableSelectionTest
13. {
14.    public static void main(String[] args)
15.    {
16.       EventQueue.invokeLater(new Runnable()
17.          {
18.             public void run()
19.             {
20.                JFrame frame = new TableSelectionFrame();
21.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22.                frame.setVisible(true);
23.             }
24.          });
25.    }
26. }
27.
28. /**
29.  * Этот фрейм показывает таблицу умножения и имеет меню для настройки
30.  * режимов выбора строк, столбцов и ячеек, а также для добавления
31.  * и удаления строк и столбцов.
32.  */
33. class TableSelectionFrame extends JFrame
34. {
35.    public TableSelectionFrame()
36.    {
37.       setTitle("TableSelectionTest");
38.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
39.
40.       // Настройка таблицы умножения.
41.
42.       model = new DefaultTableModel(10, 10);
43.
```

```
44.       for (int i = 0; i < model.getRowCount(); i++)
45.         for (int j = 0; j < model.getColumnCount(); j++)
46.           model.setValueAt((i + 1) * (j + 1), i, j);
47.
48.       table = new JTable(model);
49.
50.       add(new JScrollPane(table), "Center");
51.
52.       removedColumns = new ArrayList<TableColumn>();
53.
54.       // Создание меню.
55.       JMenuBar menuBar = new JMenuBar();
56.       setJMenuBar(menuBar);
57.
58.       JMenu selectionMenu = new JMenu("Selection");
59.       menuBar.add(selectionMenu);
60.
61.       final JCheckBoxMenuItem rowsItem = new JCheckBoxMenuItem("Rows");
62.       final JCheckBoxMenuItem columnsItem = new JCheckBoxMenuItem("Columns");
63.       final JCheckBoxMenuItem cellsItem = new JCheckBoxMenuItem("Cells");
64.
65.       rowsItem.setSelected(table.getRowSelectionAllowed());
66.       columnsItem.setSelected(table.getColumnSelectionAllowed());
67.       cellsItem.setSelected(table.getCellSelectionEnabled());
68.
69.       rowsItem.addActionListener(new ActionListener()
70.           {
71.               public void actionPerformed(ActionEvent event)
72.               {
73.                    table.clearSelection();
74.                    table.setRowSelectionAllowed(rowsItem.isSelected());
75.                    cellsItem.setSelected(table.getCellSelectionEnabled());
76.               }
77.           });
78.        selectionMenu.add(rowsItem);
79.
80.        columnsItem.addActionListener(new ActionListener()
81.           {
82.                 public void actionPerformed(ActionEvent event)
83.                 {
84.                      table.clearSelection();
85.                      table.setColumnSelectionAllowed(columnsItem.isSelected());
86.                      cellsItem.setSelected(table.getCellSelectionEnabled());
87.                  }
88.             });
89.         selectionMenu.add(columnsItem);
90.
91.         cellsItem.addActionListener(new ActionListener()
92.            {
93.               public void actionPerformed(ActionEvent event)
94.               {
95.                    table.clearSelection();
96.                    table.setCellSelectionEnabled(cellsItem.isSelected());
97.                    rowsItem.setSelected(table.getRowSelectionAllowed());
98.                    columnsItem.setSelected(table.getColumnSelectionAllowed());
99.               }
100.           });
```

```
101.        selectionMenu.add(cellsItem);
102.
103.        JMenu tableMenu = new JMenu("Edit");
104.        menuBar.add(tableMenu);
105.
106.        JMenuItem hideColumnsItem = new JMenuItem("Hide Columns");
107.        hideColumnsItem.addActionListener(new ActionListener()
108.            {
109.                public void actionPerformed(ActionEvent event)
110.                {
111.                    int[] selected = table.getSelectedColumns();
112.                    TableColumnModel columnModel = table.getColumnModel();
113.
114.                     // Удаление столбцов из представления, начиная с последнего
115.                     // индекса, чтобы не были затронуты номера столбцов
116.
117.                     for (int i = selected.length - 1; i >= 0; i--)
118.                     {
119.                       TableColumn column = columnModel.getColumn(selected[i]);
120.                       table.removeColumn(column);
121.
122.                       // Хранение удаленных столбцов
123.                       // для команды "show columns" (показать столбцы)
124.
125.                       removedColumns.add(column);
126.                     }
127.                 }
128.             });
129.          tableMenu.add(hideColumnsItem);
130.
131.          JMenuItem showColumnsItem = new JMenuItem("Show Columns");
132.          showColumnsItem.addActionListener(new ActionListener()
133.              {
134.                     public void actionPerformed(ActionEvent event)
135.                     {
136.                         // Восстановление всех удаленных столбцов
137.                         for (TableColumn tc : removedColumns)
138.                             table.addColumn(tc);
139.                         removedColumns.clear();
140.                     }
141.                 });
142.             tableMenu.add(showColumnsItem);
143.
144.             JMenuItem addRowItem = new JMenuItem("Add Row");
145.             addRowItem.addActionListener(new ActionListener()
146.                 {
147.                     public void actionPerformed(ActionEvent event)
148.                     {
149.                       // Добавление новой строки в таблицу умножения в модели
150.                       Integer[] newCells = new Integer[model.getColumnCount()];
151.                       for (int i = 0; i < newCells.length; i++)
152.                          newCells[i] = (i + 1) * (model.getRowCount() + 1);
153.                       model.addRow(newCells);
154.                     }
155.                 });
156.             tableMenu.add(addRowItem);
157.
```

```
158.             JMenuItem removeRowsItem = new JMenuItem("Remove Rows");
159.             removeRowsItem.addActionListener(new ActionListener()
160.                 {
161.                     public void actionPerformed(ActionEvent event)
162.                     {
163.                         int[] selected = table.getSelectedRows();
164.
165.                         for (int i = selected.length - 1; i >= 0; i--)
166.                             model.removeRow(selected[i]);
167.                     }
168.                 });
169.             tableMenu.add(removeRowsItem);
170.
171.             JMenuItem clearCellsItem = new JMenuItem("Clear Cells");
172.             clearCellsItem.addActionListener(new ActionListener()
173.                 {
174.                     public void actionPerformed(ActionEvent event)
175.                     {
176.                         for (int i = 0; i < table.getRowCount(); i++)
177.                         for (int j = 0; j < table.getColumnCount(); j++)
178.                             if (table.isCellSelected(i, j))
179.                               table.setValueAt(0, i, j);
180.                     }
181.                 });
182.             tableMenu.add(clearCellsItem);
183.     }
184.     private DefaultTableModel model;
185.     private JTable table;
186.     private ArrayList<TableColumn> removedColumns;
187.
188.     private static final int DEFAULT_WIDTH = 400;
189.     private static final int DEFAULT_HEIGHT = 300;
190. }
```

API javax.swing.table.TableModel 1.2

- `Class getColumnClass(int columnIndex)`

  Возвращает класс значений в указанном столбце. Эта информация используется классом отображения и редактором.

API javax.swing.JTable 1.2

- `TableColumnModel getColumnModel()`

  Возвращает модель столбца (т.е. “column model”), которая описывает размещение столбцов в таблице.

- `void setAutoResizeMode(int mode)`

  Устанавливает режим автоматического изменения размеров столбцов таблицы.

  Параметр: `mode` Одно из следующих значений:
  `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)`

  Возвращает или задает ширину зазора между ячейками смежных строк таблицы.

- `int getRowHeight()`
- `void setRowHeight(int height)`

  Возвращает или задает высоту всех строк таблицы по умолчанию.

- `int getRowHeight(int row)`
- `void setRowHeight(int row, int height)`

  Возвращает или задает высоту указанной строки таблицы.

- `ListSelectionModel getSelectionModel()`

  Возвращает модель выбора списка. Эта модель нужна, чтобы сделать выбор между строкой, столбцом и ячейкой.

- `boolean getRowSelectionAllowed()`
- `void setRowSelectionAllowed(boolean b)`

  Возвращает или задает свойство `rowSelectionAllowed`. Если оно имеет значение `true`, выбор строк будет производиться при щелчке пользователем на ячейках.

- `boolean getColumnSelectionAllowed()`
- `void setColumnSelectionAllowed(boolean b)`

  Возвращает или задает свойство `columnSelectionAllowed`. Если оно имеет значение `true`, выбор столбцов будет производиться при щелчке пользователем на ячейках.

- `boolean getCellSelectionEnabled()`

  Возвращает значение `true`, если `rowSelectionAllowed` и `columnSelectionAllowed` равны `true`.

- `void setCellSelectionEnabled(boolean b)`

  Присваивает свойствам `rowSelectionAllowed` и `columnSelectionAllowed` значение `b`.

- `void addColumn(TableColumn column)`

  Добавляет столбец как последний столбец в представлении таблицы.

- `void moveColumn(int from, int to)`

  Перемещает столбец, индексом таблицы которого является `from`, и который должен стать индексом `to`. При этом затрагивается только представление таблицы.

- `void removeColumn(TableColumn column)`

  Удаляет данный столбец из представления.

- `int convertRowIndexToModel(int index) 6`
- `int convertColumnIndexToModel(int index)`

  Возвращает индекс модели строки или столбца, имеющих данный индекс. Это значение отличается от `index`, если производится сортировка или фильтрация строк, либо если столбцы удаляются или перемещаются.

- `void setRowSorter(RowSorter<? extends TableModel> sorter)`

  Устанавливает сортировщик строки.

API javax.swing.table.TableColumnModel 1.2

- `TableColumn getColumn(int index)`

  Возвращает объект, описывающий столбец таблицы с заданным индексом.

API javax.swing.table.TableColumn 1.2

- `TableColumn(int modelColumnIndex)`

  Создает столбец таблицы для представления модели столбца с заданным индексом.

- `void setPreferredWidth(int width)`
- `void setMinWidth(int width)`
- `void setMaxWidth(int width)`

  Задает предпочтительную, минимальную и максимальную ширину столбца равной значению `width`.

- `void setWidth(int width)`

  Задает фактическую ширину столбца равной значению `width`.

- `void setResizable(boolean b)`

  Если значение параметра `b` равно `true`, размер столбца можно изменять.

API javax.swing.ListSelectionModel 1.2

- `void setSelectionMode(int mode)`

  Задает режим выбора.

  Параметр: `mode` Одно из следующих значений: `SINGLE_SELECTION`, `SINGLE_INTERVAL_SELECTION` и `MULTIPLE_INTERVAL_SELECTION`.

API javax.swing.DefaultRowSorter<M, I>6.0

- `void setComparator(int column, Comparator<?> comparator)`

  Устанавливает подборщик, который будет использоваться в данном столбце.

- `void setSortable(int column, boolean enabled)`

  Разрешает или запрещает сортировку для данного столбца.

- `void setRowFilter(RowFilter<? super M,? super I> filter)`

  Устанавливает фильтр строки.

API javax.swing.table.TableRowSorter<M extends TableModel> 6.0

- `void setStringConverter(TableStringConverter stringConverter)`

  Устанавливает преобразователь строк, используемый для сортировки и фильтрации.

API javax.swing.table.TableStringConverter<M extends TableModel> 6.0

- `abstract String toString(TableModel model, int row, int column)`

  Переопределите этот метод, чтобы преобразовать значения модели в данном месте в строке.

API javax.swing.RowFilter<M, I> 6.0

- `boolean include(RowFilter.Entry<? extends M,? extends I> entry)`

  Переопределите этот метод, чтобы определить удерживаемые строки.

- `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)`

  Возвращает фильтр, включающий строку, которая содержит одно из следующих значений: `EQUAL`, `NOT_EQUAL`, `AFTER` или `BEFORE`. Если будут даны любые индексы модели столбца, то поиск будет производиться только в этих столбцах. В противном случае, поиск будет производиться во всех столбцах. Для фильтра чисел, класс значения ячейки должен совпадать с классом `number`.

- `static <M,I> RowFilter<M,I> regexFilter(String regex, int... indices)`

  Возвращает фильтр, включающий строки, значение строки которых совпадает с данным регулярным выражением. Если будут заданы индексы модели столбца, поиск будет производиться только в этих столбцах. В противном случае поиск будет производиться во всех столбцах. Обратите внимание на совпадение строки, возвращенной методом `getStringValue` элемента `RowFilter.Entry`.

- `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)`

  Возвращает фильтр, включающий записи, которые включены во все фильтры или хотя бы в один из фильтров.

- `static <M,I> RowFilter<M,I> notFilter(RowFilter<M,I> filter)`

  Возвращает фильтр, который включает записи, не включенные в данный фильтр.

API javax.swing.RowFilter.Entry<M, I> 6.0

- `I getIdentifier()`

  Возвращает идентификатор данной записи строки.

- `M getModel()`

  Возвращает модель данной записи строки.

- `Object getValue(int index)`

  Возвращает значение, хранящееся в данном индексе данной строки.

- `int getValueCount()`

  Возвращает количество значений, хранящихся в данной строке.

- `String getStringValue()`

  Возвращает значение, хранящееся в данном индексе данной строки, преобразованное в строку. `TableRowSorter` создает записи, метод `getStringValue` которых вызывает преобразователь строк сортировщика.

Визуализация и редактирование ячеек

Как можно было видеть в разделе "Доступ к столбцам таблицы" ранее в главе, тип столбца определяет способ визуализации ячеек. Существуют визуализаторы по умолчанию для типов `Boolean` и `Icon`, которые отображают флажок опции или изображение. Для всех других типов вам необходимо устанавливать специальный визуализатор.

Визуализаторы ячеек таблицы подобны визуализаторам ячеек в списках, с которыми вы уже встречались. Они реализуют интерфейс `TableCellRenderer`, который имеет один метод:

```
Component getTableCellRendererComponent(JTable table, Object value,
boolean isSelected, boolean hasFocus, int row, int column)
```

Этот метод вызывается каждый раз при перерисовке таблицы и возвращает компонент, метод `paint()` которого используется для отображения содержимого ячейки.

В таблице, показанной на рис. 6.12, содержатся ячейки типа `Color`. Визуализатор просто возвращает панель с фоновым цветом, который является объектом цвета, хранящемся в этой ячейке. Цвет передается с помощью параметра `value`.

```
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);
   }
}
```



Рис. 6.12. Таблица с визуализаторами ячеек

Нетрудно заметить, что при получении ячейкой фокуса ввода объект отображения отображает рамку. (Для установки требуемой рамки мы используем объект `UIManager`. Значение ключа пришлось искать в исходном коде класса `DefaultTableCellRenderer`.)

Возможно, вы захотите использовать для индикации выбора ячейки цвет фона. Мы не будем давать конкретных рекомендаций, поскольку не исключено, что фон будет плохо сочетаться с остальными цветами. Способ индикации выбранной ячейки можно позаимствовать из рассмотренного ранее примера `ListRenderingTest` в листинге 6.3.



Если ваш объект отображения должен выводить строку текста или пиктограмму, можете создать его как подкласс класса `DefaultTableCellRenderer`. Тогда средства, реализованные в суперклассе, возьмут на себя ответственность за отображение выбранной ячейки и ячейки, обладающей фокусом ввода.

Следует указать таблице на необходимость использования класса отображения для всех объектов типа `Color`. Для этого предусмотрен метод `setDefaultRenderer()` класса `JTable`. Ему передаются объект `Class` и объект отображения:

```
table.setDefaultRenderer(Color.class, new ColorTableCellRenderer());
```

Теперь этот класс отображения можно использовать для всех объектов данного типа.

Если вы захотите выбрать класс отображения на основании какого-нибудь другого критерия, вам понадобится подкласс класса `JTable`, и вы должны будете переопределить метод `getCellRenderer`.

Визуализация заголовка

Чтобы отобразить изображение в заголовке, необходимо установить значение заголовка:

```
moonColumn.setHeaderValue(new ImageIcon("Moons.gif"));
```

Тем не менее, заголовок таблицы самостоятельно не может выбрать подходящий класс отображения для значения заголовка. Поэтому его необходимо установить вручную. Например, чтобы показать значок в заголовке столбца, необходимо вызвать следующий метод:

```
moonColumn.setHeaderRenderer(table.getDefaultRenderer(ImageIcon.class));
```

Редактирование ячеек

Модель таблицы должна получить сведения о том, какие ячейки можно редактировать. Для этого используется метод `isCellEditable()`. Чаще всего редактируемыми объявляются не отдельные ячейки, а целые столбцы. Например, в приведенном ниже фрагменте кода допускается 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;
}
```



В классе `AbstractTableModel` метод `isCellEditable` возвращает значение `false`, а в классе `DefaultTableModel` этот метод переопределен и по умолчанию возвращает значение `true`.

В программе, код которой представлен в листинге 6.7, можно устанавливать и снимать флажки опций в ячейках столбца `Gaseous`. Кроме того, как показано на рис. 6.13, можно также задавать нужное количество спутников в раскрывающихся списках, содержащихся в ячейках столбца `Moons`. Вы увидите, как можно использовать раскрывающийся список в качестве редактора ячейки.

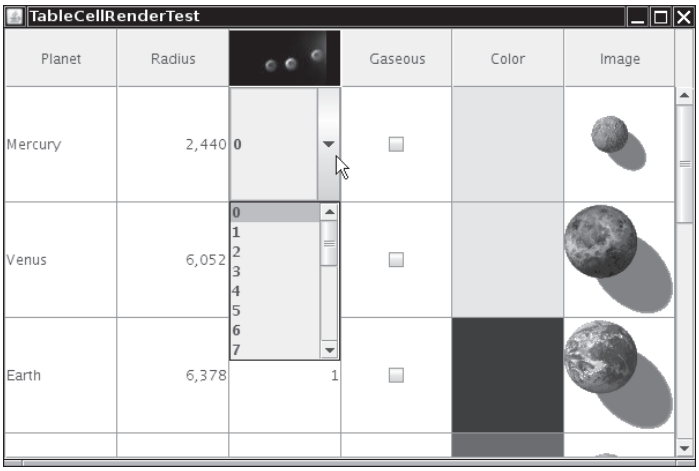


Рис. 6.13. Окно программы, позволяющей редактировать ячейки

И, наконец, в первом столбце `Planet` предусмотрено редактирование названия планеты.

В этом примере демонстрируется применение трех вариантов класса `DefaultCellEditor` для редактирования ячеек типа `JTextField`, `JCheckBox` и `JComboBox`. Класс `JTable` автоматически инсталлирует редактор флажков для ячеек типа `boolean` и поле редактирования для ячеек, содержимое которых разрешается изменять, но не определен собственный объект отображения. Поля редактирования позволяют изменять строки, которые получаются путем применения метода `toString()` к возвращаемому значению метода `getValueAt()` модели таблицы.

После завершения редактирования полученный результат извлекается путем вызова метода `getCellEditorValue()`, принадлежащего редактору. Этот метод должен возвращать значение корректного типа (т.е. типа, который возвращается методом `getColumnType()` модели).

Редактор раскрывающегося списка придется создать вручную, потому что компонент `JTable` не знает, какие значения следует использовать для такого типа ячейки. Например, в столбце `Moons` предполагается организовать выбор любого значения от 0 до 20:

```
JComboBox moonCombo = new JComboBox();
for (int i = 0; i <= 20; i++)
  moonCombo.addItem(i);
```

Чтобы создать `DefaultCellEditor` для такого типа данных, нужно передать раскрывающийся список конструктору, как показано ниже:

```
TableCellEditor moonEditor = new DefaultCellEditor(moonCombo);
```

Затем необходимо инсталлировать редактор, который, в отличие от класса отображения ячеек `Color`, не зависит от *типа* объекта. Его не обязательно использовать для всех объектов типа `Integer`, а достаточно инсталлировать только в заданном столбце:

```
moonColumn.setCellEditor(moonEditor);
```

Специальные редакторы

Запустите программу, рассматриваемую здесь в качестве примера, и щелкните кнопкой мыши на ячейке столбца `Color`. На экране появится диалоговое окно выбора цвета для данной планеты. Выберите нужный цвет, щелкните на кнопке OK, и цвет ячейки изменится требуемым образом (рис. 6.14).

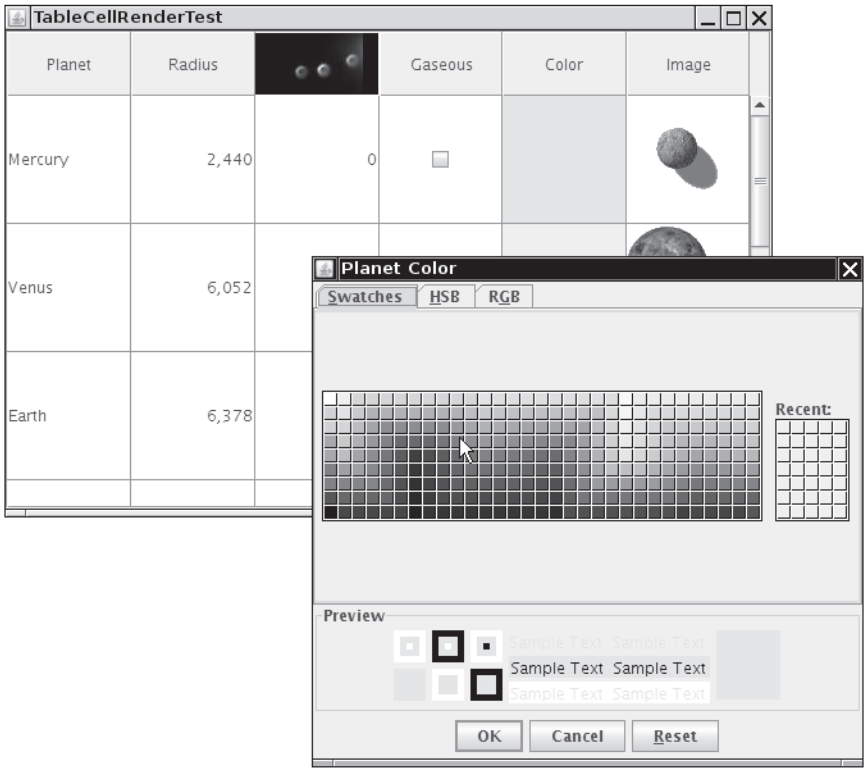


Рис. 6.14. Установка цвета ячейки

Редактор выбора цвета ячейки таблицы является не стандартным, а пользовательским, т.е. определяется разработчиком. Для создания пользовательского редактора ячеек следует применять интерфейс `TableCellEditor`. Он похож на другие интерфейсы JDK 1.3, и для обработки событий в нем используется класс `AbstractClassEditor`.

Метод `getTableCellEditorComponent()` интерфейса `TableCellEditor` запрашивает компонент для отображения ячейки. Он работает так же, как и метод `getTableCellRendererComponent()` интерфейса `TableCellRenderer`, за исключением того, что в нем нет параметра `focus`. Дело в том, что при редактировании ячейки предполагается, что эта ячейка обязательно имеет фокус ввода. На период редактирования редактор временно *заменяет* класс отображения. В нашем случае возвращается пустая панель, для которой цвет не устанавливается. Это является признаком того, что ячейка в данный момент редактируется.

Далее потребуется организовать отображение окна выбора цвета по щелчку на ячейке.

Класс JTable вызывает редактор в ответ на некоторое событие (например, щелчок кнопкой мыши). Для всех событий, которые могут инициировать редактирование, в классе AbstractCellEditor предусмотрен метод isCellEditable():

```
public boolean isCellEditable(EventObject anEvent)
{
   return true;
}
```

Однако редактор не будет использоваться в таблице, если переопределить этот метод так, чтобы он возвращал значение false.

После инсталляции редактора вызывается метод shouldSelectCell(). Если не предприняты специальные меры, он вызывается с тем же событием. В этом методе происходит инициализация процесса редактирования, например, отображается окно внешнего редактора.

```
public boolean shouldSelectCell(EventObject anEvent)
{
   colorDialog.setVisible(true);
   return true;
}
```

Для отмены или остановки редактирования (например, если пользователь щелкнул на другой ячейке) будет вызван метод cancelCellEditing() или stopCellEditing(). В обоих случаях нужно закрыть диалоговое окно. При вызове метода stopCellEditing() в таблице может появиться частично отредактированное значение. Если такое значение является допустимым, следует возвратить значение true. Любое значение, выбранное в окне выбора цвета, будет допустимым, но при редактировании других данных нужно обязательно убедиться в корректности информации, полученной из редактора.

Кроме того, следует вызвать методы суперкласса, которые позаботятся о генерировании событий; в противном случае редактирование не удастся отменить должным образом:

```
public void cancelCellEditing()
{
   colorDialog.setVisible(false);
   super.cancelCellEditing();
}
```

Наконец, потребуется определить метод получения значения, созданного в результате редактирования:

```
public Object getCellEditorValue()
{
   return colorChooser.getColor();
}
```

Итак, к пользовательскому редактору предъявляется ряд требований.

1. Он должен расширять класс AbstractCellEditor и реализовывать интерфейс TableCellEditor.
2. В нем должен быть определен метод getTableCellEditorComponent(), предназначенный для получения редактора. Компонент-редактор может отображать отдельное диалоговое окно либо находиться непосредственно в ячейке, как, например, раскрывающийся список или поле редактирования.

3. В нем должны быть определены методы `shouldSelectCell()`, `stopCellEditing()` и `cancelCellEditing()` для управления запуском, завершением и отменой процесса редактирования. Для уведомления обработчиков следует вызвать методы суперкласса `stopCellEditing()` и `cancelCellEditing()`.
4. В нем должен быть определен метод `getCellEditorValue()`, возвращающий значение, которое является результатом редактирования.

Об окончании редактирования свидетельствует вызов метода `stopCellEditing()` или `cancelCellEditing()`. При создании диалогового окна выбора цвета мы предусмотрели соответствующие методы обратного вызова.

```
colorDialog = JColorChooser.createDialog(null,
   "Planet Color", false, colorChooser,
   new
      ActionListener() // слушатель кнопки OK
      {
         public void actionPerformed(ActionEvent event)
         {
            stopCellEditing();
         }
      },
   new
      ActionListener() // слушатель кнопки Cancel
      {
         public void actionPerformed(ActionEvent event)
         {
            cancelCellEditing();
         }
      });
```

Кроме того, редактирование должно быть отменено после того, как пользователь закроет диалоговое окно. Для этого события нужно определить следующий обработчик:

```
colorDialog.addWindowListener(new
   WindowAdapter()
   {
      public void windowClosing(WindowEvent event)
      {
         cancelCellEditing();
      }
   });
```

На этом код пользовательского редактора завершен.

Теперь нам известно, как создать редактируемую ячейку и инсталлировать редактор. Остается открытым только один вопрос: как обновить модель с учетом отредактированного значения. После окончания редактирования класс `JTable` вызывает следующий метод модели таблицы:

```
void setValueAt(Object value, int r, int c);
```

Для сохранения нового значения этот метод потребуется переопределить. Параметр `value` теперь будет обозначать объект, возвращенный редактором ячейки. В определении редактора ячейки задается тип объекта, который возвращается методом `getCellEditorValue()`. В данном примере при использовании класса `DefaultCellEditor` возможны три варианта указания типа для этого значения. Для редактора флажка это тип `boolean`, для поля редактирования — `String`, а для раскрывающегося списка — тип объекта, выбранного пользователем.

Если объект value имеет другой тип, то его необходимо преобразовать к нужному типу. Чаще всего это происходит при редактировании чисел в поле редактирования. В нашем примере раскрывающийся список содержит объекты типа Integer, поэтому никакие преобразования не потребуются.

Листинг 6.7. Содержимое файла TableCellRenderTest.java

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.util.*;
4.  import javax.swing.*;
5.  import javax.swing.table.*;
6.
7.  /**
8.   * Пример перерисовки ячеек таблицы
9.   * и редактирования их значений.
10.  * @version 1.02 2007-08-01
11.  * @author Cay Horstmann
12.  */
13. public class TableCellRenderTest
14. {
15.   public static void main(String[] args)
16.   {
17.        EventQueue.invokeLater(new Runnable()
18.             {
19.                  public void run()
20.                  {
21.                       JFrame frame = new LongListFrame();
22.                       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.                       frame.setVisible(true);
24.                  }
25.             });
26.   }
27. }
28.
29. /**
30.  * Фрейм содержит таблицу с данными о планетах.
31.  */
32. class TableCellRenderFrame extends JFrame
33. {
34.   public TableCellRenderFrame()
35.   {
36.      setTitle("TableCellRenderTest");
37.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
38.
39.      TableModel model = new PlanetTableModel();
40.      JTable table = new JTable(model);
41.      table.setRowSelectionAllowed(false);
42.
43.      // Установка объектов отображения и редакторов.
44.
45.      table.setDefaultRenderer(Color.class,
46.        new ColorTableCellRenderer());
47.      table.setDefaultEditor(Color.class,
48.        new ColorTableCellEditor());
49.
50.      JComboBox moonCombo = new JComboBox();
```

```
51.       for (int i = 0; i <= 20; i++)
52.         moonCombo.addItem(i);
53.
54.       TableColumnModel columnModel = table.getColumnModel();
55.       TableColumn moonColumn = columnModel.getColumn(
56.           PlanetTableModel.MOONS_COLUMN);
57.       moonColumn.setCellEditor(new DefaultCellEditor(moonCombo));
58.       moonColumn.setHeaderRenderer(table.getDefaultRenderer
59.          (ImageIcon.class));
60.       moonColumn.setHeaderValue(new ImageIcon("Moons.gif"));
61.
62.       // Отображение таблицы.
63.
64.       table.setRowHeight(100);
65.       add(new JScrollPane(table), BorderLayout.CENTER);
66.    }
67.
68.    private static final int DEFAULT_WIDTH = 600;
69.    private static final int DEFAULT_HEIGHT = 400;
70. }
71.
72. /**
73.  * Модель таблицы определяет значения,
74.  * свойства визуализации и редактирования.
75.  */
76. class PlanetTableModel extends AbstractTableModel
77. {
78.    public String getColumnName(int c)
79.    {
80.         return columnNames[c];
81.    }
82.    public Class getColumnClass(int c)
83.    {
84.         return cells[0][c].getClass();
85.     }
86.    public int getColumnCount()
87.    {
88.         return cells[0].length;
89.    }
90.    public int getRowCount()
91.    {
92.         return cells.length;
93.    }
94.    public Object getValueAt(int r, int c)
95.    {
96.         return cells[r][c];
97.    }
98.    public void setValueAt(Object obj, int r, int c)
99.    {
100.        cells[r][c] = obj;
101.   }
102.   public boolean isCellEditable(int r, int c)
103.   {
104.        return c == PLANET_COLUMN || c == MOONS_COLUMN || c == GASEOUS_COLUMN ||
105.               c == COLOR_COLUMN;
106.   }
107.
```

```
108.    public static final int PLANET_COLUMN = 0;
109.    public static final int MOONS_COLUMN = 2;
110.    public static final int GASEOUS_COLUMN = 3;
111.    public static final int COLOR_COLUMN = 4;
112.
113.    private Object[][] cells =
114.    {
115.       { "Mercury", 2440.0,  0, false, Color.yellow,
116.          new ImageIcon("Mercury.gif") },
117.       { "Venus", 6052.0, 0, false, Color.yellow,
118.          new ImageIcon("Venus.gif") },
119.       { "Earth", 6378.0, 1, false, Color.blue,
120.          new ImageIcon("Earth.gif") },
121.       { "Mars", 3397.0, 2, false, Color.red,
122.          new ImageIcon("Mars.gif") },
123.       { "Jupiter", 71492.0, 16, true, Color.orange,
124.          new ImageIcon("Jupiter.gif") },
125.       { "Saturn", 60268.0, 18, true, Color.orange,
126.          new ImageIcon("Saturn.gif") },
127.       { "Uranus", 25559.0, 17, true, Color.blue,
128.          new ImageIcon("Uranus.gif") },
129.       { "Neptune", 24766.0, 8, true, Color.blue,
130.          new ImageIcon("Neptune.gif") },
131.       { "Pluto", 1137.0, 1, false, Color.black,
132.          new ImageIcon("Pluto.gif") }
133.    };
134.
135.    private String[] columnNames = { "Planet", "Radius", "Moons",
136.       "Gaseous", "Color", "Image" };
137. }
138.
139. /**
140.  * Объект отображения представляет значение
141.  * посредством панели соответствующего цвета.
142.  */
143. class ColorTableCellRenderer extends JPanel
144.    implements TableCellRenderer
145. {
146.    public Component getTableCellRendererComponent(JTable table,
147.       Object value, boolean isSelected, boolean hasFocus,
148.       int row, int column)
149.    {
150.       setBackground((Color) value);
151.       if (hasFocus)
152.          setBorder(UIManager.getBorder
153.             ("Table.focusCellHighlightBorder"));
154.       else
155.          setBorder(null);
156.       return this;
157.    }
158. }
159.
160. /**
161.  * Редактор отображает диалоговое окно выбора цвета.
162.  */
163. class ColorTableCellEditor extends AbstractCellEditor
164.    implements TableCellEditor
```

```
{
   public ColorTableCellEditor()
   {
      panel = new JPanel();
      // Подготовка диалогового окна выбора цвета.

      colorChooser = new JColorChooser();
      colorDialog = JColorChooser.createDialog(null,
        "Planet Color", false, colorChooser,
        new ActionListener() // слушатель кнопки OK.
          {
            public void actionPerformed(ActionEvent event)
               { stopCellEditing(); }
          }, new ActionListener() // слушатель кнопки Cancel.
          {
            public void actionPerformed(ActionEvent event)
            {
                 cancelCellEditing();
             }
          });
      colorDialog.addWindowListener(new
        WindowAdapter()
        {
          public void windowClosing(WindowEvent event)
            { cancelCellEditing(); }
        });
   }

   public Component getTableCellEditorComponent(JTable table,
     Object value, boolean isSelected, int row, int column)
   {
      // Получение текущего значения Color.
      // Если пользователь начинает редактирование,
      // это значение устанавливается в диалоговом окне.
      colorChooser.setColor((Color) value);
      return panel;
   }

   public boolean shouldSelectCell(EventObject anEvent)
   {
      // Начало редактирования.
      colorDialog.setVisible(true);

      // Вызывающий метод получает уведомление
      // о допустимости выбора данной ячейки.
      return true;
   }

   public void cancelCellEditing()
   {
      // Редактирование отменено - скрыть диалоговое окно.
      colorDialog.setVisible(false);
      super.cancelCellEditing();
   }

   public boolean stopCellEditing()
   {
```

```
222.      // Редактирование завершено - скрыть диалоговое окно.
223.      colorDialog.setVisible(false);
224.      super.stopCellEditing();
225.
226.      // Вызывающий метод уведомляется о том,
227.      // что можно использовать значение цвета.
228.      return true;
229.   }
230.
231.   public Object getCellEditorValue()
232.   {
233.      return colorChooser.getColor();
234.   }
235.
236.   private Color color;
237.   private JColorChooser colorChooser;
238.   private JDialog colorDialog;
239.   private JPanel panel;
240. }
```

API javax.swing.JTable 1.2

- `TableCellRenderer getDefaultRenderer(Class<?> type)`

  Возвращает объект отображения, используемый по умолчанию для данного типа.

- `TableCellEditor getDefaultEditor(Class<?> type)`

  Возвращает редактор, используемый по умолчанию для данного типа.

API 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 рисования ячейки таблицы.

Параметры:		
	`table`	Таблица, содержащая ячейки, которые должны быть отображены
	`value`	Отображаемая ячейка
	`selected`	Значение `true`, если текущая ячейка выбрана
	`hasFocus`	Значение `true`, если текущая ячейка содержит фокус ввода
	`row, column`	Номер строки и столбца с отображаемой ячейкой

API javax.swing.table.TableColumn 1.2

- `void setCellEditor(TableCellEditor editor)`
- `void setCellRenderer(TableCellRenderer renderer)`

  Задает редактор и класс отображения для всех ячеек заданного столбца.

- `void setHeaderRenderer(TableCellRenderer renderer)`

  Задает класс отображения для ячейки, содержащей заголовок данного столбца.

- `void setHeaderValue(Object value)`

  Задает значение, которое должно быть отображено в качестве заголовка данного столбца.

API javax.swing.DefaultCellEditor 1.2

- `DefaultCellEditor(JComboBox comboBox)`

  Создает редактор, который предоставляет для выбора значения ячейки раскрывающийся список.

API 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 рисования ячейки таблицы.

Параметры:		
	`table`	Таблица, содержащая ячейки, которые должны быть отображены
	`value`	Отображаемая ячейка
	`selected`	Значение `true`, если текущая ячейка выбрана
	`row, column`	Номер строки и столбца, на пересечении которых находится данная ячейка

API javax.swing.CellEditor 1.2

- `boolean isCellEditable(EventObject event)`

  Возвращает значение `true`, если событие может запустить процесс редактирования данной ячейки.

- `boolean shouldSelectCell(EventObject anEvent)`

  Запускает процесс редактирования. Возвращает значение `true`, если редактируемая ячейка должна быть *выделена*. Если надо, чтобы процесс редактирования не изменял выбор ячейки, следует вернуть значение `false`.

- `void cancelCellEditing()`

  Отменяет процесс редактирования. Результаты можно игнорировать.

- `boolean stopCellEditing()`

  Останавливает процесс редактирования. Предполагается, что частичные результаты будут использованы. Возвращает значение `true`, если значение, полученное в результате редактирования, является допустимым.

- `Object getCellEditorValue()`

  Возвращает результаты редактирования.

- `void addCellEditorListener(CellEditorListener l)`
- `void removeCellEditorListener(CellEditorListener l)`

  Включает и удаляет обязательный обработчик события редактирования ячейки.

Деревья

Каждому пользователю компьютера, в котором используется иерархическая файловая система, знакомы древовидные представления файлов и каталогов. Понятно, что это всего лишь один из многих примеров применения древовидной структуры. В повседневной жизни такую же структуру образует система территориально-административного деления на страны, штаты, области и города (рис. 6.15).

Программистам довольно часто приходится создавать код для отображения подобных структур. Для этого в библиотеке Swing предусмотрен класс `JTree`. Вместе со вспомогательными классами (которые предназначены для выполнения рутинных стандартных действий) он создает структуру дерева и обрабатывает запросы пользователей на развертывание и свертывание узлов. В этом разделе обсуждается использование класса `JTree`.

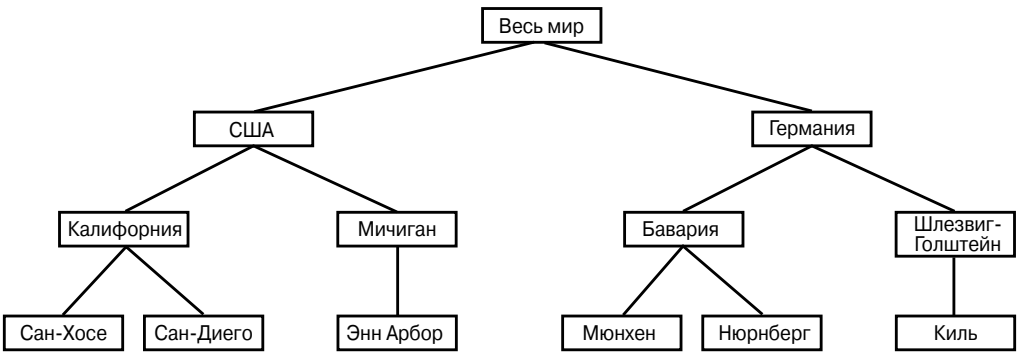


Рис. 6.15. Древовидная система территориально-административного деления на страны, штаты, области и города

Как и для других сложных компонентов Swing, здесь рассматриваются только самые общие и наиболее распространенные приемы использования класса `JTree` без описания деталей. Более подробное описание данного класса можно найти в книге Дэвида М. Гери (David M. Geary) *Graphic Java 2: Mastering the JFC, Volume II: Swing, 3rd Edition*, (Prentice-Hall, 1999 г.) либо в книгах Кима Топли (Kim Topley) *Core Java Foundation Classes* (Prentice-Hall, 1998 г.) и *Core Swing: Advanced Programming* (Pearson Education 1999 г.).

Для изучения приведенного в данном разделе материала познакомимся с некоторыми основными терминами (рис. 6.16). Дерево состоит из *узлов* (node). Каждый узел может либо быть *листовым* (leaf), либо иметь *дочерние узлы* (child nodes). Каждый узел, за исключением корневого, имеет только одного *родителя* (parent). Дерево имеет только один *корневой узел* (root node). Набор деревьев, каждое из которых имеет свой собственный корень, называется *лесом* (forest).

Простые деревья

В первом простом примере попытаемся создать дерево с несколькими узлами (рис. 6.18). Как и с большинством других компонентов библиотеки Swing, программист создает модель иерархических данных, а компонент автоматически отображает их. Для создания объекта `JTree` нужно предоставить конструктору модель дерева:

```
TreeModel model = ...;
JTree tree = new JTree(model);
```

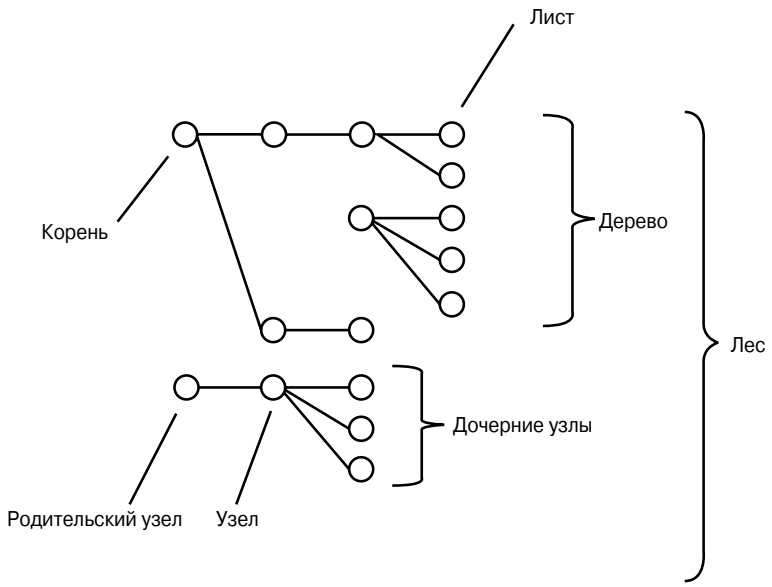


Рис. 6.16. Терминология, используемая для описания древовидных структур



Некоторые конструкторы создают деревья на основе набора элементов:

```
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 только один узел. Третий конструктор и вовсе бесполезен, поскольку узлы появляются в нем в произвольном порядке, который задается хеш-кодами ключей.

Как же получить модель дерева? Можно создать собственную модель с помощью класса, реализующего интерфейс `TreeModel`. Этот способ будет рассмотрен далее в главе, а сейчас попробуем использовать предлагаемую в библиотеке Swing модель `DefaultTreeModel`.

Для формирования модели дерева по умолчанию нужно предоставить конструктору корневой узел:

```
TreeNode root = ...;
DefaultTreeModel model = new DefaultTreeModel(root);
```

Здесь `TreeNode` также представляет собой интерфейс. В используемую по умолчанию модель можно вставлять экземпляры любого класса, реализующего этот интерфейс. В данном примере применяется уже готовый класс-узел `DefaultMutableTreeNode` из библиотеки Swing. Как показано на рис. 6.17, этот класс реализует интерфейс `MutableTreeNode`, который является дочерним по отношению к интерфейсу `TreeNode`.

Экземпляр класса `DefaultMutableTreeNode` содержит *пользовательский объект* узла дерева. Эти объекты отображаются для всех узлов. Если не задан специальный объект отображения, то будет выведена строка, полученная с помощью метода `toString()`.

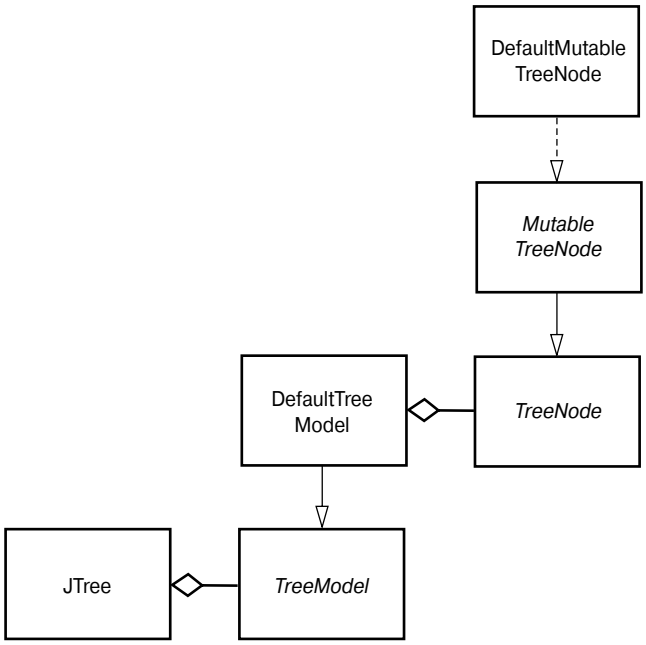


Рис. 6.17. Схема наследования интерфейсов, используемых при создании деревьев

В первом примере в качестве пользовательских объектов применяются строки, хотя на практике обычно используются более сложные объекты. Например, для отображения дерева каталогов имеет смысл использовать объекты `File`.

Пользовательский объект можно указать либо в конструкторе, либо позже с помощью метода `setUserObject()`:

```
DefaultMutableTreeNode node = new DefaultMutableTreeNode("Texas");
node.setUserObject("California");
```

Затем между узлами следует установить отношение "родительский–дочерний". Для этого нужно, начиная с корневого узла, добавлять с помощью метода `add()` дочерние узлы:

```
DefaultMutableTreeNode root = new DefaultMutableTreeNode("World");
DefaultMutableTreeNode country =
   new DefaultMutableTreeNode("USA");
root.add(country);
DefaultMutableTreeNode state = new
   DefaultMutableTreeNode("California");
country.add(state);
```

Полученное дерево показано на рис. 6.18.

Подобным образом следует связать все узлы, а затем создать класс-модель `DeafultTreeNode` с корневым узлом и на основе этой модели сам компонент-дерево `JTree`:

```
DefaultTreeModel treeModel = new
DefaultTreeModel(root);
JTree tree = new JTree(treeModel);
```



Рис. 6.18. Простая древовидная структура

Можно поступить еще проще: передать корневой узел конструктору класса `JTree`, который автоматически создаст принимаемую по умолчанию модель дерева:

```
JTree tree = new JTree(root);
```

Полный код программы приведен в листинге 6.8.

Листинг 6.8. Содержимое файла `SimpleTree.java`

```
1.  import java.awt.*;
2.
3.  import javax.swing.*;
4.  import javax.swing.tree.*;
5.
6.  /**
7.   * Отображение простой древовидной структуры.
8.   * @version 1.02 2007-08-01
9.   * @author Cay Horstmann
10.  */
11. public class SimpleTree
12. {
13.   public static void main(String[] args)
14.   {
15.        EventQueue.invokeLater(new Runnable()
16.             {
17.                  public void run()
18.                  {
19.                      JFrame frame = new LongListFrame();
20.                      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.                      frame.setVisible(true);
22.                  }
23.              });
24.   }
25. }
26.
27. /**
28.  * Этот фрейм содержит простое дерево, которое отображает
29.  * созданную вручную модель.
30.  */
31. class SimpleTreeFrame extends JFrame
32. {
33.   public SimpleTreeFrame()
34.   {
35.      setTitle("SimpleTree");
36.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
37.
38.      // Установка данных для модели дерева.
39.
40.      DefaultMutableTreeNode root = new DefaultMutableTreeNode("World");
41.      DefaultMutableTreeNode country = new DefaultMutableTreeNode("USA");
42.      root.add(country);
43.      DefaultMutableTreeNode state = new DefaultMutableTreeNode("California");
44.      country.add(state);
45.      DefaultMutableTreeNode city = new DefaultMutableTreeNode("San Jose");
46.      state.add(city);
47.      city = new DefaultMutableTreeNode("Cupertino");
48.      state.add(city);
49.      state = new DefaultMutableTreeNode("Michigan");
50.      country.add(state);
```

```
51.       city = new DefaultMutableTreeNode("Ann Arbor");
52.       state.add(city);
53.       country = new DefaultMutableTreeNode("Germany");
54.       root.add(country);
55.       state = new DefaultMutableTreeNode("Schleswig-Holstein");
56.       country.add(state);
57.       city = new DefaultMutableTreeNode("Kiel");
58.       state.add(city);
59.
60.       // Формирование дерева и включение его
61.       // в состав панели с возможностью прокрутки.
62.       JTree tree = new JTree(root);
63.       add(new JScrollPane(tree));
64.    }
65.    private static final int DEFAULT_WIDTH = 300;
66.    private static final int DEFAULT_HEIGHT = 200;
67. }
```

Дерево, отображаемое в результате выполнения данной программы, будет выглядеть так, как показано на рис. 6.19. В окне программы будут видны только корневой узел и его дочерние узлы. Для развертывания поддеревьев нужно щелкнуть мышью на пиктограмме в виде круга (маркере узла). Штрих, соединяющийся с кругом, направлен вправо, если поддерево свернуто, или вниз, если оно развернуто (рис. 6.20). Неизвестно, что имели в виду разработчики стиля Metal, но, видимо, изображение круга со штрихом является стилизованным обозначением дверной ручки, которую нужно повернуть вниз по часовой стрелке, чтобы "открыть" поддерево.

Конечно, внешний вид дерева зависит от выбранного программистом стиля. Приведенное выше описание относится к стилю Metal. В стилях Windows и Motif для обозначения свернутого и развернутого дерева используются символы + и - (рис. 6.21).



Рис. 6.19. Исходное состояние отображаемого дерева

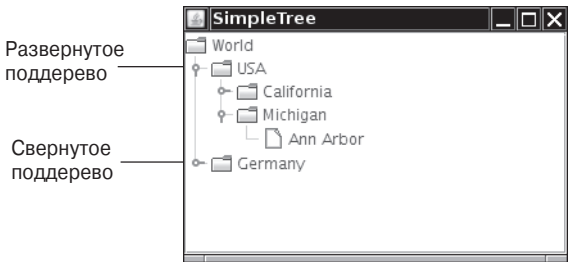


Рис. 6.20. Свернутое и развернутое поддерево



Рис. 6.21. Внешний вид дерева, отображаемого с использованием стиля Windows

Чтобы скрыть линии, связывающие родительские и дочерние узлы, нужно задать значение `None` свойства `JTree.lineStyle` (рис. 6.22):

```
tree.putCLientProperty("JTree.lineStyle", "None");
```

И, наоборот, для отображения этих линий нужно задать значение `Angled`:

```
tree.putCLientProperty("JTree.lineStyle", "Angled");
```

На рис. 6.23 показан еще один стиль представления структуры дерева с помощью горизонтальных линий `Horizontal`. Это дерево содержит горизонтальные линии, которые разделяют только дочерние узлы корневого узла. На первый взгляд трудно представить себе ситуацию, когда могла бы пригодиться такая структура.

По умолчанию корневой узел не содержит маркера для свертывания всего дерева. Если этот маркер необходим, следует вызвать представленный ниже метод:

```
tree.setShowsRootHandles(true);
```



Рис. 6.22. Дерево без соединительных линий между узлами

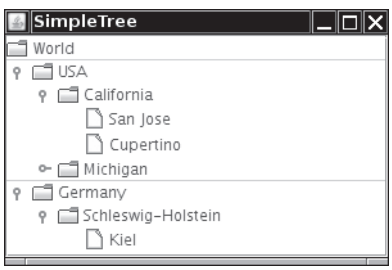


Рис. 6.23. Представление структуры дерева с помощью горизонтальных линий

На рис. 6.24 показан результат вызова данного метода. Теперь все дерево можно сворачивать и разворачивать целиком.

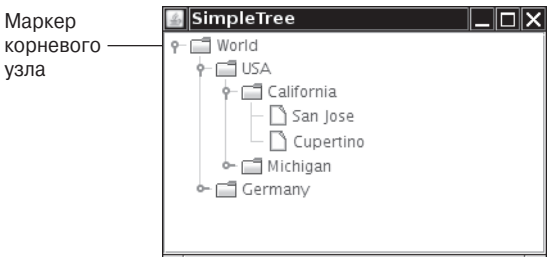


Рис. 6.24. Дерево с маркером корневого узла

Кроме того, корневой узел можно скрыть. Это может пригодиться, например, для отображения *леса*, т.е. набора 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`, которые объединены в одном дереве со скрытым корневым узлом.



Рис. 6.25. Отображение леса

Рассмотрим теперь листья дерева. Для их отображения используется пиктограмма, показанная на рис. 6.26.

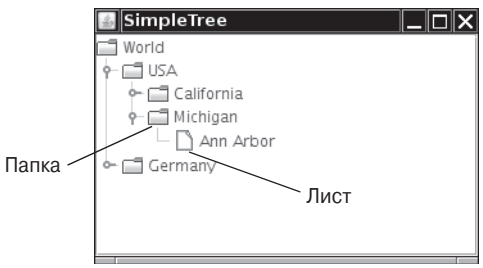


Рис. 6.26. Пиктограммы для отображения узлов различных типов

Существуют три типа пиктограмм, применяемых для представления листьев дерева: в виде листа бумаги, а также в виде открытой и закрытой папки.

Объект отображения узла должен знать, какой тип пиктограммы нужно применять в каждом конкретном случае. По умолчанию решение принимается следующим образом: если метод `isLeaf()` возвращает значение `true`, то используется пиктограмма в виде листа бумаги, в противном случае — пиктограмма папки.

Метод `isLeaf()` класса `DefaultMutableTreeNode` возвращает значение `true`, если данный узел не имеет дочерних узлов. Таким образом, узлы с дочерними узлами будут показаны в виде папок, а узлы без них — в виде бумажных листов.

Такой способ обозначения узлов подходит не для всех случаев. Например, при добавлении узла `Montana` для обозначения штата Монтана без указания городов, этот штат будет обозначен пиктограммой в виде листа бумаги. Однако при этом будет нарушена логическая концепция представления дерева, в которой такие пиктограммы используются для представления городов.

Класс `JTree` не имеет информации о том, какие узлы являются листами. Для ответа на этот вопрос он использует модель. Если узел без дочерних узлов с концептуальной точки зрения не является листом, можно применять другой критерий определения листов, например, на основе допустимости дочерних узлов.

Если для некоторого узла дочерние узлы недопустимы, нужно вызвать следующий метод:

```
node.setAllowsChildren(false);
```

Затем потребуется указать модели на то, что для определения, является ли узел листом, необходимо проверять свойство, соответствующее допустимости дочерних узлов. Для организации упомянутого поведения нужно применить метод `setAsksAllowsChildren()` класса `DefaultTreeModel`:

```
model.setAsksAllowsChildren(true);
```

В таком случае узлы, для которых допускается наличие дочерних узлов, будут обозначены пиктограммами-папками, а узлы, для которых не допускается наличие дочерних узлов, будут обозначены пиктограммами-листами.

Если в конструкторе дерева вы задаете корневой узел, вы также можете указать в нем на необходимость проверки допустимости дочерних узлов:

```
JTree tree = new JTree(root, tree);
// Узлы, для которых не допускается наличие дочерних
// узлов, будут обозначены пиктограммами-листами.
```

API javax.swing.JTree 1.2

- `JTree(TreeModel model)`

  Создает дерево на основе указанной модели.

- `JTree(TreeNode root)`
- `JTree(TreeNode root, boolean asksAllowChildren)`

  Создает дерево с заданной по умолчанию моделью, отображающее корневой узел и его дочерние узлы.

Параметры:		
	`root`	Корневой узел
	`asksAllowChildren`	Значение `true` указывает на то, что для определения, является ли узел листом, необходимо проверить допустимость наличия дочерних узлов

- `void setShowsRootHandles(boolean b)`

  Если параметр `b` имеет значение `true`, для корневого узла отображается маркер свертывания.

- `void setRootVisible(boolean b)`

  Если параметр `b` имеет значение `true`, корневой узел отображается.

API javax.swing.tree.TreeNode 1.2

- `boolean isLeaf()`

  Возвращает значение `true`, если данный узел является листом.

- `boolean getAllowsChildren()`

  Возвращает значение `true`, если данный узел может иметь дочерние узлы.

API javax.swing.tree.MutableTreeNode 1.2

- `void setUserObject(Object userObject)`

  Задает пользовательский объект `userObject`, который используется для отображения узла дерева.

API javax.swing.tree.TreeModel 1.2

- `boolean isLeaf(Object node)`

  Возвращает значение `true`, если узел `node` следует отобразить как лист.

API javax.swing.tree.DefaultTreeModel 1.2

- `void setAsksAllowsChildren(boolean b)`

  Если параметр `b` имеет значение `true`, то узел отображается как лист при условии, что метод `getAllowsChildren()` возвращает значение `false`. В противном случае такой внешний вид узла будет выбираться, если метод `isLeaf()` возвращает значение `true`.

API javax.swing.tree.DefaultMutableTreeNode 1.2

- `DefaultMutableTreeNode(Object userObject)`

  Создает узел для заданного пользовательского объекта.

- `void add(MutableTreeNode child)`

  Вставляет узел в дерево в виде последнего дочернего узла для данного узла.

- `void setAllowsChildren(boolean b)`

  Если параметр `b` имеет значение `true`, к данному узлу могут быть добавлены дочерние узлы.

API javax.swing.JComponent 1.2

- `void putClientProperty(Object key, Object value)`

  Создает пару ключ/значение и включает ее в небольшую таблицу, которой управляет каждый компонент. Этот механизм используется в некоторых компонентах библиотеки Swing для хранения специальных свойств, управляющих их внешним видом.

Редактирование деревьев и путей к деревьям

В следующем примере рассматриваются способы редактирования деревьев. На рис. 6.27 показан пользовательский интерфейс программы, в котором для создания нового узла с именем `New` предусмотрены кнопки Add Sibling (Добавить равноправный) и Add Child (Добавить дочерний). Для удаления текущего выбранного узла предназначена кнопка Delete (Удалить).



Рис. 6.27. Окно программы, предназначенной для редактирования дерева

Для реализации такого поведения нужно найти текущий выбранный узел. В классе `JTree` предусмотрен интересный способ поиска такого узла на основе *пути к объекту* (path of object). Путь начинается с корневого узла и состоит из последовательности дочерних узлов; пример пути показан на рис. 6.28.

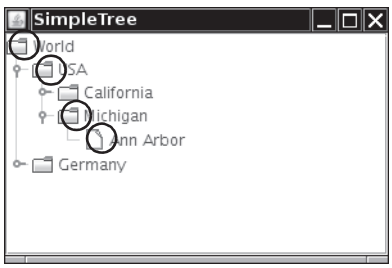


Рис. 6.28. Пример пути к объекту

Казалось бы, что для поиска текущего узла лучше было использовать интерфейс `TreeNode` и метод `getParent()`. Но на самом деле класс `JTree` ничего не знает об интерфейсе `TreeNode`, поскольку он используется при реализации класса `DefaultTreeModel`, а не в интерфейсе `TreeModel`. Дерево может иметь модель, в которой не используется 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, а класс `JTree` не имеет никаких сведений о них. Поэтому в классе `JTree` предлагается использовать путь к объекту.

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 использовать метод `getLastPathComponent()`. Для поиска текущего выбранного узла применяется метод `getSelectionPath()` класса `JTree`. Таким образом, зная путь к объекту, т.е. на основании объекта `TreePath`, можно получить ссылку на текущий выбранный узел:

```
TreePath selectionPath = tree.getSelectionPath();
DefaultMutableTreeNode selectedNode =
   (DefaultMutableTreeNode)selectionPath.getLastPathComponent();
```

Эта задача выполняется очень часто, поэтому для нее придуман специальный метод:

```
DefaultMutableTreeNode selectedNode =
   (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
```

Этот метод называется `getLastSelectedPathComponent()`, а не `getSelectedNode()`, поскольку дереву ничего не известно об узлах, а модель дерева работает только с путями.



Кроме путей к объектам, для описания узлов дерева используются методы класса `JTree`, которые работают с целочисленными индексами — *позициями строк* (row positions). Позиция строки — это просто номер строки (нумерация начинается с нуля) узла во внутреннем представлении дерева. Такие номера имеют только видимые узлы, причем номер строки может меняться при свертывании, развертывании или изменении дерева. Поэтому для доступа к узлам не рекомендуется применять номера строк. Каждому методу класса `JTree`, предназначенному для работы с номерами строк, соответствует метод на основе путей, который выполняет те же действия.

После получения выбранного узла можно приступать к его редактированию. Однако для создания дочернего узла нельзя просто применить метод `add()`:

```
selctedNode.add(newNode); // Так делать нельзя!
```

При изменении структуры узлов изменяется модель дерева, а не связанное с ним представление. Следовательно, нужно послать соответствующее уведомление об этом. Если для вставки нового узла использовать метод `insertNodeInto()` класса `DefaultTreeModel`, то такое уведомление будет отправлено 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 с сохранением структуры узла нужно использовать метод `nodeChanged()`:

```
model.nodeChanged(changedNode);
```

Автоматическое уведомление является основным достоинством модели `DeafultTreeModel`. При создании собственных моделей дерева потребуется самостоятельно организовать такое уведомление. Более подробное обсуждение этой темы можно найти в книге Кима Топли (Kim Topley) *Core Java Foundation Classes*.



Класс `DeafultTreeModel` содержит метод `reload()`, который перезагружает всю модель. Однако не следует вызывать метод `reload()` лишь для обновления дерева после внесения нескольких изменений. Дело в том, что при такой регенерации дерева все узлы за пределами дочерних узлов корневого узла снова будут свернуты. Такое поведение дерева может помешать нормальной работе пользователей.

Если представление получает уведомление об изменении структуры узлов, оно будет обновлено, но не развернуто для просмотра добавленных узлов. Например, добавление нового дочернего узла для свернутых подчиненных деревьев будет незаметно

для пользователя. В таком случае придется специально организовать развертывание родительских узлов для отображения вставленного нового дочернего узла. Для этого можно применить метод `makeVisible()` класса `JTree`, который использует путь, ведущий к отображаемому на экране узлу.

Таким образом, для отображения вновь вставленного узла потребуется создать путь от корневого узла. Для получения этого пути сначала следует вызвать метод `getPathToRoot()` класса `DeafultTreeModel`. Он возвращает массив `TreeArray[]` для всех узлов — с текущего до корневого, который передается конструктору `TreePath`.

Ниже показано, как выглядит фрагмент кода для отображения вновь вставленного узла.

```
TreeNode[] nodes = model.getPathToRoot(newNode);
TreePath path = new TreePath(nodes);
tree.makeVisible(path);
```



Как ни странно, класс `DefaultTreeModel` ведет себя так, будто ему ничего не известно о классе `TreePath`, хотя его работа заключается именно во взаимодействии с классом `JTree`. В своей работе класс `JTree` всегда использует пути и никогда не применяет массивы узлов.

Предположим, что дерево находится в панели с полосой прокрутки. 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()` так, как показано ниже.

```
tree.scrollPathToVisible(path);
```

Этот метод развертывает все узлы в составе пути и прокручивает содержимое окна вплоть до последнего узла в конце пути (рис. 6.29).

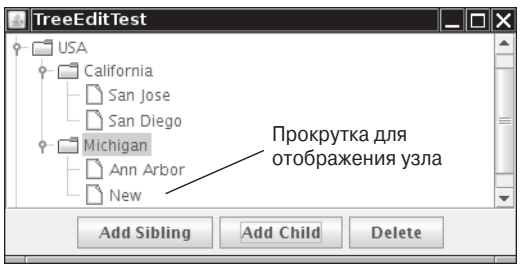


Рис. 6.29. Автоматическая прокрутка дерева для просмотра нового узла

По умолчанию узлы дерева нельзя редактировать, но с помощью метода `setEditable()` можно отменить эту установку:

```
tree.setEditable(true);
```

В таком случае после двойного щелчка мышью откроется показанное на рис. 6.30 *окно редактора ячейки* (default cell editor). Для его создания используется класс `DefaultCellEditor`. По окончании редактирования строки следует нажать клавишу <Enter>. Для редактирования объектов-узлов можно использовать собственные редакторы ячеек (о них идет речь в разделе, посвященном таблицам, где они применяются наиболее часто).



Рис. 6.30. Пример использования редактора ячейки, предлагаемого по умолчанию

В листинге 6.9 представлен полный код программы редактирования узлов дерева. Запустите программу, 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 6.9. Содержимое файла `TreeEditTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import javax.swing.*;
4.  import javax.swing.tree.*;
5.
6.  /**
7.   * Пример редактирования дерева.
8.   * @version 1.03 2007-08-01
9.   * @author Cay Horstmann
10.  */
11. public class TreeEditTest
12. {
13.    public static void main(String[] args)
14.    {
15.         EventQueue.invokeLater(new Runnable()
16.              {
17.                   public void run()
18.                   {
19.                        JFrame frame = new TreeEditFrame();
20.                        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.                        frame.setVisible(true);
22.                   }
23.              });
24.    }
25. }
26. /**
27.  * Фрейм с деревом и кнопками для его редактирования.
28.  */
29. class TreeEditFrame extends JFrame
30. {
31.    public TreeEditFrame()
32.    {
33.       setTitle("TreeEditTest");
34.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
35.
36.       // Создание дерева.
```

```
37.       TreeNode root = makeSampleTree();
38.       model = new DefaultTreeModel(root);
39.       tree = new JTree(model);
40.       tree.setEditable(true);
41.
42.       // Помещение дерева в состав панели с полосой прокрутки.
43.
44.       JScrollPane scrollPane = new JScrollPane(tree);
45.       add(scrollPane, BorderLayout.CENTER);
46.
47.       makeButtons();
48.    }
49.
50.    public TreeNode makeSampleTree()
51.    {
52.       DefaultMutableTreeNode root =
53.          new DefaultMutableTreeNode("World");
54.       DefaultMutableTreeNode country =
55.          new DefaultMutableTreeNode("USA");
56.       root.add(country);
57.       DefaultMutableTreeNode state =
58.          new DefaultMutableTreeNode("California");
59.       country.add(state);
60.       DefaultMutableTreeNode city =
61.          new DefaultMutableTreeNode("San Jose");
62.       state.add(city);
63.       city = new DefaultMutableTreeNode("San Diego");
64.       state.add(city);
65.       state = new DefaultMutableTreeNode("Michigan");
66.       country.add(state);
67.       city = new DefaultMutableTreeNode("Ann Arbor");
68.       state.add(city);
69.       country = new DefaultMutableTreeNode("Germany");
70.       root.add(country);
71.       state = new DefaultMutableTreeNode("Schleswig-Holstein");
72.       country.add(state);
73.       city = new DefaultMutableTreeNode("Kiel");
74.       state.add(city);
75.       return root;
76.    }
77.
78.    /**
79.     * Создание кнопок для вставки равноправного
80.     * и дочернего узлов, а также для удаления узла.
81.     */
82.    public void makeButtons()
83.    {
84.       JPanel panel = new JPanel();
85.       JButton addSiblingButton = new JButton("Add Sibling");
86.       addSiblingButton.addActionListener(new ActionListener()
87.          {
88.             public void actionPerformed(ActionEvent event)
89.             {
90.                DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode)
91.                   tree.getLastSelectedPathComponent();
92.
93.                if (selectedNode == null) return;
```

```
94.             DefaultMutableTreeNode parent = (DefaultMutableTreeNode)
95.               selectedNode.getParent();
96.
97.             if (parent == null) return;
98.
99.             DefaultMutableTreeNode newNode =
100.              new DefaultMutableTreeNode("New");
101.
102.            int selectedIndex = parent.getIndex(selectedNode);
103.            model.insertNodeInto(newNode, parent,
104.              selectedIndex + 1);
105.
106.            // Отображение нового узла.
107.
108.            TreeNode[] nodes = model.getPathToRoot(newNode);
109.            TreePath path = new TreePath(nodes);
110.            tree.scrollPathToVisible(path);
111.         }
112.       });
113.    panel.add(addSiblingButton);
114.
115.    JButton addChildButton = new JButton("Add Child");
116.    addChildButton.addActionListener(new ActionListener()
117.       {
118.          public void actionPerformed(ActionEvent event)
119.          {
120.             DefaultMutableTreeNode selectedNode =
121.                (DefaultMutableTreeNode)
122.                tree.getLastSelectedPathComponent();
123.
124.             if (selectedNode == null) return;
125.
126.             DefaultMutableTreeNode newNode =
127.               new DefaultMutableTreeNode("New");
128.             model.insertNodeInto(newNode, selectedNode,
129.               selectedNode.getChildCount());
130.
131.             // Отображение нового узла.
132.
133.             TreeNode[] nodes = model.getPathToRoot(newNode);
134.             TreePath path = new TreePath(nodes);
135.             tree.scrollPathToVisible(path);
136.          }
137.       });
138.    panel.add(addChildButton);
139.
140.    JButton deleteButton = new JButton("Delete");
141.    deleteButton.addActionListener(new ActionListener()
142.       {
143.          public void actionPerformed(ActionEvent event)
144.          {
145.             DefaultMutableTreeNode selectedNode =
146.               (DefaultMutableTreeNode) tree.getLastSelectedPathComponent();
147.
148.             if (selectedNode != null && selectedNode.getParent() != null)
149.               model.removeNodeFromParent(selectedNode);
150.          }
151.       });
```

```
152.       panel.add(deleteButton);
153.       add(panel, BorderLayout.SOUTH);
154.    }
155.
156.    private DefaultTreeModel model;
157.    private JTree tree;
158.    private static final int DEFAULT_WIDTH = 400;
159.    private static final int DEFAULT_HEIGHT = 200;
160. }
```

API javax.swing.JTree 1.2

- `TreePath getSelectionPath()`

  Определяет путь к текущему выбранному узлу (или путь к первому выбранному узлу, если выбрано сразу несколько узлов). Возвращает `null`, если никакой узел не выбран.

- `Object getLastSelectedPathComponent()`

  Определяет объект, который представляет текущий выбранный узел (или первый выбранный узел, если выбрано сразу несколько узлов). Возвращает `null`, если никакой узел не выбран.

- `void makeVisible(TreePath path)`

  Развертывает все узлы вдоль заданного пути.

- `void scrollPathToVisible((TreePath path)`

  Развертывает все узлы вдоль указанного пути и, если дерево содержится в панели с полосой прокрутки, прокручивает дерево, чтобы обеспечить отображение последнего узла пути.

API javax.swing.tree.TreePath 1.2

- `Object getLastPathComponent`

  Возвращает последний объект на заданном пути.

API javax.swing.tree.TreeNode 1.2

- `TreeNode getParent()`

  Возвращает родительский узел для заданного узла.

- `TreeNode getChildAt(int index)`

  Возвращает дочерний узел, соответствующий заданному индексу. Значение индекса должно находиться в интервале от `0` до `getChildCount()-1`.

- `int getChildCount()`

  Возвращает количество дочерних узлов для данного узла.

- `Enumeration children()`

  Возвращает объект `Enumeration`, который обеспечивает перебор всех дочерних узлов для данного узла.

API javax.swing.tree.DefaultTreeModel 1.2

- `void insertNodeInto(MutableTreeNode newChild, MutableTreeNode parent, int index)`

  Вставляет объект `neweChild` в качестве дочернего узла для родительского узла `parent`. Положение нового узла среди дочерних узлов определяется индексом `index`.

- `void removeNodeFromParent(MutableTreeNode node)`

  Удаляет узел `node` из модели и модифицирует обработчики модели.

- `void nodeChanged(TreeNode node)`

  Уведомляет обработчиков модели о том, что узел `node` изменился.

- `void nodesChanged(TreeNode parent, int[], changedChildIndexes)`

  Уведомляет обработчиков модели о том, что изменились дочерние узлы узла `parent` с заданными индексами.

- `void reload()`

  Перезагружает все узлы в модели. Эту операцию нужно выполнять только в тех случаях, когда узлы полностью изменились в результате внешнего воздействия.

Перечисление узлов

Иногда требуется найти узел дерева, начиная с корневого узла и перебирая все дочерние. Класс `DefaultMutableTreeNode` предлагает несколько удобных методов для перебора узлов.

Методы `breadthFirstEnumeration()` и `depthFirstEnumeration()` возвращают объект `Enumeration`. Этот объект содержит метод `nextElement()`, предназначенный для последовательного извлечения содержимого. Если объект `Enumeration` получен с помощью метода `breadthFirstEnumeration()`, он представляет результаты обхода в ширину (breadth-first) всех дочерних узлов текущего узла. Объект `Enumeration`, возвращенный методом `depthFirstEnumeration()` содержит результаты обхода в глубину (depth-first). На рис. 6.31 условно показаны оба способа обхода узлов.

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. Результаты такого обхода отображаются достаточно просто.

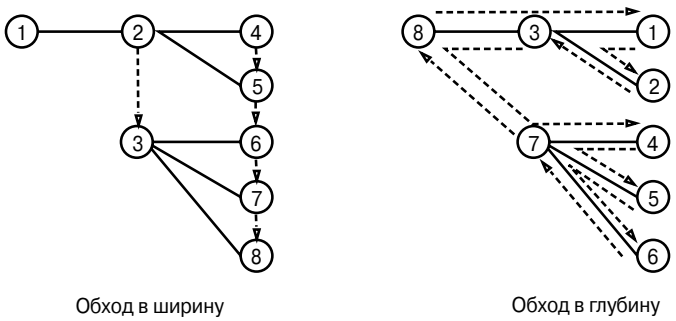


Рис. 6.31. Способы обхода узлов дерева

Обход узлов дерева в глубину похож на поиск выхода из лабиринта. В этом случае обход выполняется по какому-то одному пути вплоть до достижения узла-листа. Затем после возврата на некоторое количество уровней находится ближайший новый путь, и обход продолжается по этому пути вплоть до достижения следующего узла-листа и т.д.

Такой обход дерева называется также *обходом в обратном порядке* (postorder traversal), поскольку в процессе поиска сначала посещаются дочерние, а потом родительские узлы. Метод `postOrderTraversal()` является синонимом метода `depthFirstTraversal()`. Следует отметить также метод `preOrderTraversal()`. Он тоже предназначен для обхода в глубину, но в наборе результатов родительские узлы предшествуют дочерним.

Ниже показан типичный способ применения метода обхода:

```
Enumeration breadthFirst = node.breadthFirstEnumeration();
while (breadthFirst.hasMoreElements())
  действия с результатами breadthFirst.nextElement();
```

И, наконец, метод `pathFromAncestorEnumeration()` предназначен для поиска пути от родительского узла к заданному и для перечисления узлов вдоль этого пути. Принцип его работы основан 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 рассматривается применение описанных выше методов для обхода дерева наследования классов. Введите имя класса в поле редактирования в нижней части окна программы. После этого в дерево наследования будет добавлен указанный класс и все его суперклассы, как показано на рис. 6.32.

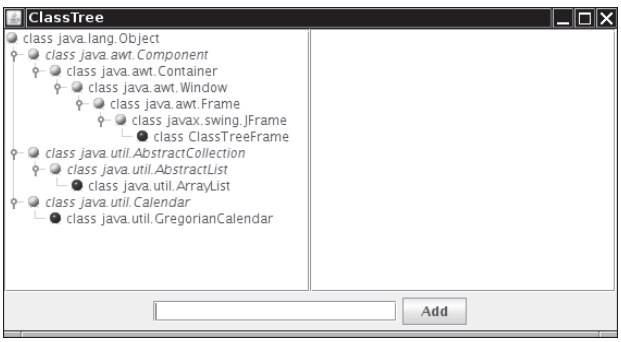


Рис. 6.32. Пример дерева наследования классов

В этом примере используется важная особенность дерева, состоящая в том, что пользовательский объект узла может быть объектом произвольного типа. В примере узлы представляют иерархическую структуру классов, поэтому будут иметь тип `Class`.

Во избежание дублирования нужно организовать проверку наличия включаемого класса в дереве. Это осуществляется 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;
   }
```

Визуализация узлов

В приложениях часто возникает необходимость изменить способ отображения узлов, например, применять другие пиктограммы для узлов или другой шрифт для их названий. Эти изменения можно вносить с помощью *объекта отображения узлов дерева* (tree cell renderer). По умолчанию для рисования узлов в классе `JTree` используется класс `DefaultTreeCellRenderer`, который расширяет класс `JLabel`. Данный класс формирует пиктограмму и метку узла.



Объект отображения узлов дерева не выводит маркеры для свертывания/развертывания подчиненных деревьев. Они являются частью общего стиля, и их не рекомендуется изменять.

Настройка внешнего вида узлов может быть выполнена одним из трех способов.

- Замена с помощью класса `DefaultTreeCellRenderer` пиктограммы, шрифта и цвета фона, которые применяются для всех узлов дерева.
- Расширение используемого по умолчанию объекта отображения узлов дерева `DefaultTreeCellRenderer` и использование разных пиктограмм, шрифта и цвета фона для разных узлов дерева.
- Установка объекта отображения узлов, который реализует интерфейс `TreeCellRenderer`. Этот объект может выводить для разных узлов дерева изображения, предусмотренные разработчиком.

Рассмотрим более подробно каждый из этих способов. Наиболее простой способ изменения пиктограммы, шрифта и цвета фона узлов основан на применении заданного по умолчанию объекта отображения узлов дерева `DefaultTreeCellRenderer`:

```
DefaultTreeCellRenderer renderer = new DefaultTreeCellRenderer();
renderer.setLeafIcon(new ImageIcon("blue-ball.gif"));
   // Для листьев
renderer.setClosedIcon(new ImageIcon("red-ball.gif"));
   // Для свернутых узлов
renderer.setOpenIcon(new ImageIcon("yellow-ball.gif"));
   // Для развернутых узлов
tree.setCellRenderer(renderer);
```

Результат выполнения этих действий показан на рис. 6.32, где в качестве пиктограмм узлов используются изображения шариков.

Не рекомендуется изменять шрифт и цвет фона, потому что при этом легко нарушить общий стиль интерфейса приложения. Изменять шрифт допустимо лишь для выделения отдельных узлов. Например, на рис. 6.32 курсивом выделены абстрактные классы.

Для управления внешним видом отдельных узлов нужно создать собственный объект отображения. Этот объект во многом напоминает объект отображения ячеек списка, рассмотренный ранее в главе. Интерфейс `TreeCellRenderer` имеет единственный метод `getTreeCellRendererComponent()`:

```
Component getTreeCellRendererComponent(JTree tree, Object value,
   boolean selected, boolean expanded, boolean leaf, int row,
   boolean hasFocus)
```

Метод `getTreeCellRendererComponent()` класса `DefaultTreeCellRenderer` возвращает значение `this`, т.е. данную надпись. (Напомним, что класс `DefaultTreeCellRenderer` расширяет класс `JLabel`.) Для настройки этого компонента следует создать подкласс класса `DefaultTreeCellRenderer`.

Затем нужно переопределить метод `getTreeCellRendererComponent()`, предусмотрев в нем вызов метода суперкласса, который подготовит все данные, необходимые для создания метки. Кроме того, в методе задаются нужные значения свойств, и по окончании выполнения он возвращает значение `this`.

```
class MyTreeCellRenderer extends DefaultTreeCellRenderer
{
  public Component getTreeCellRendererComponent(JTree tree,
    Object value, boolean selected,  boolean expanded,
    boolean leaf, int row, boolean hasFocus)
  {
    super.getTreeCellRendererComponent(tree, value,
      selected, expanded, leaf, row, hasFocus);
    DefaultMutableTreeNode node =
      (DefaultMutableTreeNode) value;
    пользовательский объект node.getUserObject();
    Font font = требуемый шрифт;
    setFont(font);
    return this;
  }
};
```



Параметр `value` метода `getTreeCellRendererComponent()` является объектом-*узлом*, *а не* пользовательским объектом. Напомним, что пользовательский объект является компонентом `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 сделано в предыдущем примере.



В классе `DefaultTreeCellRenderer` используется *одинаковый* объект-метка для всех узлов, но изменяется текст для каждого узла. Если вы изменили шрифт для отдельного узла, при следующем вызове этого метода нужно восстановить используемый по умолчанию шрифт. В противном случае все следующие узлы будут использовать новый шрифт! Способ восстановления исходного шрифта показан в листинге 6.10.

Здесь не будет рассматриваться пример использования объекта отображения для вывода произвольной пиктограммы. Если вам потребуется сделать это, вы без труда адаптируете для своей задачи объект отображения ячеек списка, показанный в листинге 6.3.

`ClassNameTreeCellRenderer` в листинге 6.10 отображает имя класса обычным шрифтом или курсивом, в зависимости от наличия модификатора `ABSTRACT` у объекта `Class`. Для обозначения абстрактных классов не используется никакой другой шрифт, чтобы не изменять стиль, обычно применяемый для отображения дерева. По этой причине курсив для абстрактного класса *создается* путем наклона базового шрифта. Учтите, что существует только один совместно используемый объект типа `JLabel`, который возвращается всеми вызовами. При последующих вызовах метода `getTreeCellRendererComponent` нужно снова вернуться к исходному шрифту.

Обратите внимание на способ изменения пиктограмм узлов в конструкторе `ClassTreeFrame`.

API javax.swing.tree.DefaultMutableTreeNode 1.2

- `Enumeration breadthFirstEnumeration()`
- `Enumeration depthFirstEnumeration()`
- `Enumeration preOrderEnumeration()`
- `Enumeration postOrderEnumeration()`

Возвращает объект `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 синонимом метода `depthFirstEnumeration()`. Метод `preOrderEnumeration()` выполняет те же действия, что и `postOrderEnumeration`, только родительские узлы учитываются раньше дочерних.

API 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 отображает узел дерева.

Параметры:		
	`tree`	Дерево, которое содержит узел, предназначенный для отображения
	`value`	Отображаемый узел
	`selected`	Имеет значение `true`, если данный узел выбран
	`expanded`	Имеет значение `true`, если показаны дочерние узлы данного узла
	`leaf`	Имеет значение `true`, если данный узел нужно представить как лист
	`row`	Строка, содержащая узел
	`hasFocus`	Имеет значение `true`, если текущий узел обладает фокусом ввода

API javax.swing.tree.DefaultTreeCellRenderer 1.2

- `void setLeafIcon(Icon icon)`
- `void setOpenIcon(Icon icon)`
- `void setClosedIcon(Icon icon)`

Задает пиктограмму для обозначения листа, развернутого узла и свернутого узла.

Обработка событий деревьев

Дерево чаще всего используется совместно с каким-то другим компонентом. Например, при выборе какого-то узла в соседнем окне может быть показана какая-то сопроводительная информация. На рис. 6.33 представлен пример программы, в которой для каждого узла в правом текстовом окне автоматически будут показаны статические переменные и переменные экземпляра соответствующего класса.

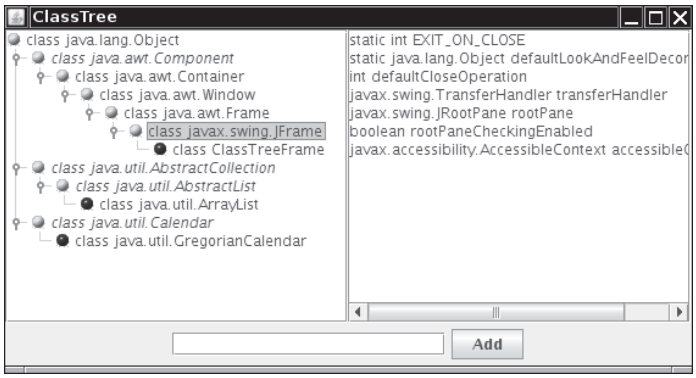


Рис. 6.33. Пример программы для просмотра классов

Для создания такой программы нужно использовать *обработчик событий выбора узлов дерева* (tree selection listener). Класс обработчика должен реализовать интерфейс `TreeSelectionListener` с единственным методом `valueChanged()`:

```
void valueChanged(TreeSelection event);
```

Метод `valueChanged()` вызывается при каждом событии выбора или отмены выбора узлов дерева.

Обработчик подключается к дереву стандартным образом:

```
tree.addTreeSelectionListener(listener);
```

При этом можно с помощью свойств модели выбора `TreeSelectionModel` специально указать, какой способ выбора узлов дерева учитывается: 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 рассматриваемой здесь программе просмотра классов допускается выбор только одного узла.

```
int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
tree.getSelectionModel().setSelectionMode(mode);
```

Кроме указания режима выбора, не нужно предпринимать никаких других действий.



Способ выбора нескольких элементов зависит от стиля отображения. При работе со стилем Metal для выбора несмежных узлов следует, удерживая нажатой клавишу <Ctrl>, щелкать на каждом выбираемом элементе, а для выбора нескольких *смежных* узлов необходимо, удерживая нажатой клавишу <Shift>, щелкнуть на первом и на последнем элементах выбираемой группы.

Для определения текущего выбора нужно запросить дерево, вызывая метод `getSelectionPaths()`:

```
TreePath[] selectedPaths = getSelectionPaths();
```

Если вы ограничите возможности пользователя и разрешите выбирать только один узел, удобнее использовать метод `getSelectionPath()`, который в общем случае возвращает первый выбранный путь или значение `null`, если элемент не выбран.



Класс `TreeSelectionEvent` имеет метод `getPaths()`, который возвращает массив объектов `TreePath`, но этот массив описывает изменения выбора, а не текущий выбор.

В листинге 6.10 приведен весь исходный код программы, отображающей иерархию наследования; курсивом выводятся названия абстрактных классов. После ввода полного имени класса и нажатия клавиши <Enter> или щелчка на кнопке Add (Добавить) в дерево будет вставлен новый класс со всеми его суперклассами. Вы должны ввести полное имя пакета, такое как `java.util.ArrayList`.

Программа несколько сложна для понимания, поскольку в ней для формирования дерева классов используется механизм рефлексии. Весь код создания дерева классов находится в методе `addClass()`. (Структура классов используется здесь лишь как удобный пример дерева с иерархией наследования. В собственных приложениях программисты могут использовать любые другие источники иерархических данных.) Для проверки наличия вставляемого класса применяется метод `findUserObject()`, реализация которого описана в предыдущем разделе. Он реализует алгоритм обхода в ширину. Если класса нет в данном дереве, то вставляется его суперкласс, а затем создается и отображается новый узел-класс.

При выделении узла дерева область текста справа заполняется полями выбранного класса. В конструкторе фрейма мы ограничиваем пользователя одним выбором элемента и добавляем слушатель (обработчик) операции выбора в дереве. При вызове метода `valueChanged()` игнорируется значение его параметра и лишь запрашивается у дерева текущий выбранный путь. Далее, как обычно, нужно получить последний узел пути и извлечь его пользовательский объект. Затем для создания строки со всеми полями выбранного класса используется метод `getFieldDescription()`.

Листинг 6.10. Содержимое файла `ClassTree.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.lang.reflect.*;
4.  import java.util.*;
5.  import javax.swing.*;
6.  import javax.swing.event.*;
7.  import javax.swing.tree.*;
8.
9.  /**
10.  * Программа, иллюстрирующая процесс отображения узлов
11.  * дерева, представляющих классы и их суперклассы.
12.  * @version 1.03 2007-08-01
13.  * @author Cay Horstmann
14.  */
15. public class ClassTree
16. {
17.    public static void main(String[] args)
18.    {
19.         EventQueue.invokeLater(new Runnable()
20.               {
21.                    public void run()
22.                    {
23.                        JFrame frame = new ClassTreeFrame();
24.                        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25.                        frame.setVisible(true);
26.                    }
27.               });
28.    }
29.
```

```
30. /**
31.  * Данный фрейм содержит дерево классов, поле редактирования
32.  * и кнопку для вставки новых классов.
33.  */
34. class ClassTreeFrame extends JFrame
35. {
36.    public ClassTreeFrame()
37.    {
38.       setTitle("ClassTree");
39.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
40.
41.       // Корневым узлом дерева классов является узел типа Object.
42.       root = new DefaultMutableTreeNode(java.lang.Object.class);
43.       model = new DefaultTreeModel(root);
44.       tree = new JTree(model);
45.
46.       // Включение класса для заполнения дерева данными.
47.       addClass(getClass());
48.
49.       // Установка пиктограмм узлов.
50.       ClassNameTreeCellRenderer renderer = new ClassNameTreeCellRenderer();
51.       renderer.setClosedIcon(new ImageIcon("red-ball.gif"));
52.       renderer.setOpenIcon(new ImageIcon("yellow-ball.gif"));
53.       renderer.setLeafIcon(new ImageIcon("blue-ball.gif"));
54.       tree.setCellRenderer(renderer);
55.
56.       // Установка режима выбора.
57.       tree.addTreeSelectionListener(new TreeSelectionListener()
58.       {
59.            public void valueChanged(TreeSelectionEvent event)
60.            {
61.                 // Пользователь выбирает другой режим — обновляем описание.
62.                 TreePath path = tree.getSelectionPath();
63.                  if (path == null) return;
64.                  DefaultMutableTreeNode selectedNode =
65.                    (DefaultMutableTreeNode) path .getLastPathComponent();
66.                  Class<?> c = (Class<?>) selectedNode.getUserObject();
67.                  String description = getFieldDescription(c);
68.                  textArea.setText(description);
69.              }
70.          });
71.       int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
72.       tree.getSelectionModel().setSelectionMode(mode);
73.
74.       // Эта область текста содержит описание класса.
75.       textArea = new JTextArea();
76.
77.       // Добавление дерева и текстовой области.
78.       JPanel panel = new JPanel();
79.       panel.setLayout(new GridLayout(1, 2));
80.       panel.add(new JScrollPane(tree));
81.       panel.add(new JScrollPane(textArea));
82.
83.       add(panel, BorderLayout.CENTER);
84.
85.       addTextField();
86. }
```

```
87.  /**
88.   * Добавление поля редактирования и кнопки Add для включения нового класса.
89.   */
90.    public void addTextField()
91.    {
92.       JPanel panel = new JPanel();
93.
94.       ActionListener addListener = newActionListener()
95.          {
96.             public void actionPerformed(ActionEvent event)
97.             {
98.                // Включение класса, имя которого
99.                // задано в поле редактирования.
100.               try
101.               {
102.                  String text = textField.getText();
103.                  addClass(Class.forName(text)); // Очищение поля редактирования
104.                                                 // свидетельствует об удачном выполнении.
105.                  textField.setText("");
106.               }
107.               catch (ClassNotFoundException e)
108.               {
109.                  JOptionPane.showMessageDialog(null, "Class not found");
110.               }
111.            }
112.         };
113.
114.      // В данном поле вводятся имена новых классов.
115.      textField = new JTextField(20);
116.      textField.addActionListener(addListener);
117.      panel.add(textField);
118.
119.      JButton addButton = new JButton("Add");
120.      addButton.addActionListener(addListener);
121.      panel.add(addButton);
122.
123.      add(panel, BorderLayout.SOUTH);
124.   }
125.
126.   /**
127.    * Поиск объекта в дереве.
128.    * @param obj Искомый объект
129.    * @return Узел, содержащий объект, или null,
130.    *   если объект отсутствует
131.    */
132.   @SuppressWarnings("unchecked")
133.   public DefaultMutableTreeNode findUserObject(Object obj)
134.   {
135.      // Поиск узла, содержащего пользовательский объект.
136.      Enumeration e = root.breadthFirstEnumeration();
137.      while (e.hasMoreElements())
138.      {
139.         DefaultMutableTreeNode node = (DefaultMutableTreeNode) e.nextElement();
140.         if (node.getUserObject().equals(obj)) return node;
141.      }
142.      return null;
143.   }
```

```
144.   /**
145.    * Добавление нового класса и родительских
146.    * классов, еще не являющихся частью дерева.
147.    * @param c Добавляемый класс
148.    * @return Добавленный узел
149.    */
150.   public DefaultMutableTreeNode addClass(Class c)
151.   {
152.      // Включение нового класса в состав дерева.
153.
154.      // Пропуск типов, не являющихся классами.
155.      if (c.isInterface() || c.isPrimitive()) return null;
156.
157.      // Если класс уже присутствует в составе дерева,
158.      // возвращается соответствующий узел.
159.      DefaultMutableTreeNode node = findUserObject(c);
160.      if (node != null) return node;
161.
162.      // Класс отсутствует - рекурсивное включение родительских классов.
163.
164.      Class s = c.getSuperclass();
165.
166.      DefaultMutableTreeNode parent;
167.      if (s == null) parent = root;
168.      else parent = addClass(s);
169.
170.      // Добавление класса в качестве дочернего.
171.      DefaultMutableTreeNode newNode = new DefaultMutableTreeNode(c);
172.      model.insertNodeInto(newNode, parent, parent.getChildCount());
173.
174.      // Визуализация узла.
175.      TreePath path = new TreePath(model.getPathToRoot(newNode));
176.      tree.makeVisible(path);
177.
178.      return newNode;
179.   }
180.
181.   /**
182.    * Возвращает описание полей класса.
183.    * @param описываемый класс
184.    * @return строка, содержащая все типы полей и имена
185.    */
186.   public static String getFieldDescription(Class<?> c)
187.  {
188.        // Используем рефлексию для поиска типов и имен полей.
189.        StringBuilder r = new StringBuilder();
190.        Field[] fields = c.getDeclaredFields();
191.        for (int i = 0; i < fields.length; i++)
192.        {
193.           Field f = fields[i];
194.            if ((f.getModifiers() & Modifier.STATIC) != 0) r.append("static ");
195.            r.append(f.getType().getName());
196.            r.append(" ");
197.            r.append(f.getName());
198.            r.append("\n");
199.        }
200.        return r.toString();
201.   }
```

```
202.    private DefaultMutableTreeNode root;
203.    private DefaultTreeModel model;
204.    private JTree tree;
205.    private JTextField textField;
206.    private static final int DEFAULT_WIDTH = 400;
207.    private static final int DEFAULT_HEIGHT = 300;
208. }
209.
210. /**
211.  * Данный класс отображает имя класса либо обычным шрифтом,
212.  * либо курсивом. Курсивом выводятся абстрактные классы.
213.  */
214. class ClassNameTreeCellRenderer extends DefaultTreeCellRenderer
215. {
216.    public Component getTreeCellRendererComponent(JTree tree, Object value,
217.     boolean selected, boolean expanded, boolean leaf, int row, boolean hasFocus)
218.    {
219.       super.getTreeCellRendererComponent(tree, value, selected, expanded,
220.                                          leaf, row, hasFocus);
221.       // Получение пользовательского объекта.
222.       DefaultMutableTreeNode node = (DefaultMutableTreeNode) value;
223.       Class c = (Class) node.getUserObject();
224.
225.       // Курсив формируется из обычного шрифта.
226.       if (plainFont == null)
227.       {
228.          plainFont = getFont();
229.          // Объект отображения узла иногда вызывается при отсутствующем шрифте.
230.
231.          if (plainFont != null) italicFont = plainFont.deriveFont(Font.ITALIC);
232.       }
233.
234.       // Если класс абстрактный, устанавливается курсив,
235.       // в противном случае - обычный шрифт.
236.       if ((c.getModifiers() & Modifier.ABSTRACT) == 0) setFont(plainFont);
237.       else setFont(italicFont);
238.       return this;
239.    }
240.
241.    private Font plainFont = null;
242.    private Font italicFont = null;
243. }
```

API javax.swing.JTree 1.2

- `TreePath getSelectionPath()`
- `TreePath[] getSelectionPaths()`

  Возвращает первый выбранный путь или массив путей ко всем выбранным узлам. При отсутствии выбора возвращается значение `null`.

API javax.swing.event.TreeSelectionListener 1.2

- `void valueChanged(TreeSelectionEvent event)`

  Вызывается при каждом выборе или отмене выбора узлов дерева.

API javax.swing.event.TreeSelectionEvent 1.2

- `TreePath getPath()`
- `TreePath[] getPaths()`

  Возвращает первый путь или все пути, которые *изменились* в результате данного события. Для получения сведений о текущем выборе, а не изменении выбора следует вызывать метод `JTree.getSelectionPath()`.

Специальные модели деревьев

В качестве последнего примера реализуем программу, которая, подобно отладчику, проверяет содержимое переменной. Окно программы показано на рис. 6.34.



Рис. 6.34. Пример дерева с функциями проверки

Скомпилируйте и запустите эту программу, прежде чем читать описание принципов ее работы. Каждый узел дерева соответствует какой-то переменной экземпляра. Если переменная является объектом, то разверните структуру этого объекта для просмотра *его* переменных экземпляра. Как видите, эта программа позволяет просматривать содержимое своего фрейма. При более внимательном изучении нескольких переменных экземпляра можно найти некоторые уже знакомые нам классы. Кроме того, эта программа позволяет оценить, насколько сложны компоненты библиотеки Swing, предназначенные для организации пользовательского интерфейса.

Особенностью программы является то, что для дерева не используется класс модели `DefaultTreeModel`. При наличии данных, которые уже иерархически организованы в древовидной структуре, вряд ли имеет смысл использовать предлагаемое по умолчанию дерево и синхронизировать оба дерева. Именно такая ситуация имеет место в данном примере: объекты уже связаны друг с другом с помощью ссылок, а потому не нужно дублировать эту структуру связей.

Ряд удобных методов объявлено в интерфейсе `TreeModel`. Первая группа методов позволяет найти узлы дерева, начиная с корня с последующим обходом дочерних узлов. Класс `JTree` вызывает перечисленные ниже методы, только если узел был развернут:

```
Object getRoot()
int getChildCount(Object parent)
Object getChild(Object parent, int index)
```

В рассматриваемом здесь примере показано, почему для интерфейса `TreeModel`, как и для класса `JTree`, не требуется явное определение узлов. Как видите, корневые и дочерние узлы могут быть объектами произвольного типа. Интерфейс `TreeModel` отвечает за уведомление класса `JTree` об организации связи между ними.

Следующий метод выполняет действия, обратные методу `getChild()`:

```
int getIndexOfChild(Object parent, Object child);
```

Как показано в листинге 6.11, этот метод можно реализовать на основе первых трех.

Модель дерева сообщает классу дерева `JTree` о том, какие узлы следует отобразить как листья:

```
boolean isLeaf(Object node);
```

Если в результате выполнения кода изменяется модель дерева, то дерево нужно уведомить о необходимости перерисовки. Поэтому дерево добавляется в модель как обработчик `TreeModelListener`. Таким образом, модель должна использовать стандартные методы управления обработчиками:

```
void addTreeModelListener(TreeModelListener l);
void removeTreeModelListener(TreeModelListener l);
```

В листинге 6.11 показана реализация этих методов.

Для изменения содержимого дерева модель использует один из четырех перечисленных ниже методов интерфейса `TreeModelListener`:

```
void treeNodesChanged(TreeModelEvent e);
void treeNodesInserted(TreeModelEvent e);
void treeNodesRemoved(TreeModelEvent e);
void treeStructureChanged(TreeModelEvent e);
```

Объект `TreeModelEvent` описывает расположение изменения. Подробности организации событий модели дерева для обслуживания вставки или удаления узлов обсуждаться здесь не будут. Программисту достаточно позаботиться о генерировании таких событий. В листинге 6.11 приведен пример генерирования события для замены корневого узла новым объектом.



Разработчики библиотеки Swing предусмотрели специальный класс `javax.swing.EventListenerList`, который содержит набор обработчиков событий. Более подробные сведения об этом можно найти в главе 8 первого тома.

Наконец, если пользователь редактирует узел дерева, то модель вызывается с помощью следующего метода:

```
void valueForPathChanged(TreePath path, Object newValue);
```

Если редактирование не допускается, этот метод не потребуется. В таком случае модель дерева создается гораздо проще. Нужно реализовать следующие три метода:

```
Object getRoot();
int getChildCount(Object parent);
Object getChild(Object parent, int index);
```

Эти методы описывают структуру дерева. После реализации остальных пяти методов так, как показано в листинге 6.11, можно приступать к отображению дерева.

В приведенном примере программы используется дерево с объектами типа `Variable`.



Если бы в этом примере использовалась модель дерева `DefaultTreeModel`, то узлы были бы объектами типа `DefaultMutableTreeNode` с *пользовательскими объектами* типа `Variable`.

Например, переменная с объявлением `Employee joe;` имеет *тип* `Employee.class` и *имя* `joe`, а ее *значением* является ссылка на объект `joe`. Тогда класс `v` типа `Variable`, который описывает данную переменную, в этом примере будет иметь следующее объявление:

```
Variable v = new Variable(Employee.class, "joe", joe);
```

Если переменная относится к простому типу, то для работы с ее значением нужно создать объект-оболочку:

```
Variable v = new Variable(double.class, "salary", new Double(salary));
```

Если переменная является классом, то она имеет *поля*, которые можно перечислить и собрать в объекте `ArrayList`. Поскольку метод `getFields()` класса `Class` не возвращает поля суперкласса, то придется вызвать этот метод для родительского класса. Соответствующий код находится в конструкторе класса `Variable`. Метод `getFields()` класса `Variable` возвращает массив полей. Наконец, метод `toString()` класса `Variable` форматирует метку узла. Метка всегда содержит тип и имя переменной, а если переменная не является классом, то приводится ее значение.



Значения элементов массива в данном примере не отображаются. Работу с массивами реализовать нетрудно, поэтому читателю предлагается самостоятельно решить эту задачу.

Перейдем к модели дерева. Код первых двух методов прост:

```
public Object getRoot()
{
   return root;
}

public int getChildCount(Object parent)
{
   return ((Variable)parent).getFields().size();
}
```

Метод `getChild()` возвращает новый объект типа `Variable`, который описывает поле с определенным индексом. Методы `getType()` и `getName()` класса `Field` позволяют получить тип и имя поля. С помощью механизма рефлексии можно, вызывая метод `f.get(parentValue)`, прочитать значение поля. Этот метод может сгенерировать исключение `IllegalAccessException`. Однако свободный доступ ко всем полям предоставлен в конструкторе `Variable`, поэтому такая ситуация не может возникнуть.

Вот как выглядит 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;
      }
   }
```

Рассмотренные три метода описывают структуру дерева объектов для компонента `JTree`, а остальные методы выполняют другие рутинные операции. Они представлены в листинге 6.11.

Необходимо отметить следующую особенность модели: она описывает *бесконечное* дерево. Это можно проверить, рассматривая объект `WeakReference`. Щелкая на переменной `referent`, можно вернуться к исходному объекту, который содержит идентичное подчиненное дерево с объектом `WeakReference`. Конечно, не существует никакого способа хранения бесконечного количества узлов. Просто эта модель дерева может генерировать узлы по требованию по мере развертывания родительских узлов.

Данным примером мы завершаем обсуждение структуры деревьев и методов работы с ними. В следующем разделе приводится описание таблиц — еще одного сложного компонента библиотеки Swing. Хотя деревья и таблицы имеют много различий, в них используются одинаковые принципы создания модели данных и перерисовки элементов.

Листинг 6.11. Содержимое файла `ObjectInspectorTest.java`

```
1.  import java.awt.*;
2.  import java.lang.reflect.*;
3.  import java.util.*;
4.  import javax.swing.*;
5.  import javax.swing.event.*;
6.  import javax.swing.tree.*;
7.
8.  /**
9.   * Пример применения пользовательской модели
10.  * дерева для отображения полей объекта.
11.  * @version 1.23 2007-08-01
12.  * @author Cay Horstmann
13.  */
14. public class ObjectInspectorTest
15. {
16.    public static void main(String[] args)
17.    {
18.         EventQueue.invokeLater(new Runnable()
19.              {
20.                   public void run()
21.                   {
22.                        JFrame frame = new ObjectInspectorFrame();
23.                        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24.                        frame.setVisible(true);
25.                   }
26.              });
27.    }
28. }
29.
30. /**
31.  *  Фрейм, содержащий дерево объектов.
32.  */
33. class ObjectInspectorFrame extends JFrame
34. {
```

```
35.    public ObjectInspectorFrame()
36.    {
37.       setTitle("ObjectInspectorTest");
38.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
39.
40.       // Проверка фрейма с объектами.
41.
42.       Variable v = new Variable(getClass(), "this", this);
43.       ObjectTreeModel model = new ObjectTreeModel();
44.       model.setRoot(v);
45.
46.       // Формирование и отображение дерева.
47.
48.       tree = new JTree(model);
49.       add(new JScrollPane(tree), BorderLayout.CENTER);
50.    }
51.
52.    private JTree tree;
53.    private static final int DEFAULT_WIDTH = 400;
54.    private static final int DEFAULT_HEIGHT = 300;
55. }
56.
57. /**
58.  * Модель описывает структуру дерева с Java-объектами.
59.  * Дочерние объекты отображают переменные экземпляра
60.  */
61. class ObjectTreeModel implements TreeModel
62. {
63.    /**
64.     * Формирование пустого дерева.
65.     */
66.    public ObjectTreeModel()
67.    {
68.       root = null;
69.    }
70.
71.    /**
72.     * Корневой узел для данной переменной.
73.     * @param v Переменная, описываемая деревом
74.     */
75.    public void setRoot(Variable v)
76.    {
77.       Variable oldRoot = v;
78.       root = v;
79.       fireTreeStructureChanged(oldRoot);
80.    }
81.
82.    public Object getRoot()
83.    {
84.       return root;
85.    }
86.
87.    public int getChildCount(Object parent)
88.    {
89.       return ((Variable) parent).getFields().size();
90.    }
91.
```

```
92.    public Object getChild(Object parent, int index)
93.    {
94.       ArrayList<Field> fields = ((Variable) parent).getFields();
95.       Field f = (Field) fields.get(index);
96.       Object parentValue = ((Variable) parent).getValue();
97.       try
98.       {
99.          return new Variable(f.getType(), f.getName(),
100.            f.get(parentValue));
101.      }
102.      catch (IllegalAccessException e)
103.      {
104.         return null;
105.      }
106.   }
107.
108.   public int getIndexOfChild(Object parent, Object child)
109.   {
110.      int n = getChildCount(parent);
111.      for (int i = 0; i < n; i++)
112.         if (getChild(parent, i).equals(child))
113.            return i;
114.      return -1;
115.   }
116.
117.   public boolean isLeaf(Object node)
118.   {
119.      return getChildCount(node) == 0;
120.   }
121.
122.   public void valueForPathChanged(TreePath path, Object newValue)
123.   {
124.   }
125.
126.   public void addTreeModelListener(TreeModelListener l)
127.   {
128.      listenerList.add(TreeModelListener.class, l);
129.   }
130.
131.   public void removeTreeModelListener(TreeModelListener l)
132.   {
133.      listenerList.remove(TreeModelListener.class, l);
134.   }
135.
136.   protected void fireTreeStructureChanged(Object oldRoot)
137.   {
138.      TreeModelEvent event = new TreeModelEvent(this,
139.         new Object[] {oldRoot});
140.      EventListener[] listeners =
141.         listenerList.getListeners(TreeModelListener.class);
142.      for (int i = 0; i < listeners.length; i++)
143.         ((TreeModelListener)
144.         listeners[i]).treeStructureChanged(event);
145.   }
146.
147.   private Variable root;
148.   private EventListenerList listenerList = new EventListenerList();
149. }
```

```
150. /**
151.  * Переменная с указанием типа, имени и значения.
152.  */
153. class Variable
154. {
155.    /**
156.     * Формирование переменной.
157.     * @param aType Тип.
158.     * @param aName Имя.
159.     * @param aValue Значение.
160.     */
161.    public Variable(Class aType, String aName, Object aValue)
162.    {
163.       type = aType;
164.       name = aName;
165.       value = aValue;
166.       fields = new ArrayList<Field>();
167.       // Поиск всех полей класса,
168.       // за исключением строк и значений null.
169.       if (!type.isPrimitive() && !type.isArray() &&
170.           !type.equals(String.class) && value != null)
171.       {
172.          // Получение полей класса и всех суперклассов.
173.          for (Class c = value.getClass();
174.              c != null; c = c.getSuperclass())
175.          {
176.             Field[] fs = c.getDeclaredFields();
177.             AccessibleObject.setAccessible(fs, true);
178.
179.             // Получение всех нестатических полей.
180.             for (Field f : fs)
181.                if ((f.getModifiers() & Modifier.STATIC) == 0)
182.                   fields.add(f);
183.          }
184.       }
185.    }
186.    /**
187.     * Получение значения переменной.
188.     * @return Значение
189.     */
190.    public Object getValue() { return value; }
191.    /**
192.     * Получение всех нестатических полей для переменной.
193.     * @return Списочный массив для полей
194.     */
195.    public ArrayList<Field> getFields()
196.    {
197.       return fields;
198.     }
199.    public String toString()
200.    {
201.       String r = type + " " + name;
202.       if (type.isPrimitive()) r += "=" + value;
203.       else if (type.equals(String.class)) r += "=" + value;
204.       else if (value == null) r += "=null";
205.       return r;
206.    }
```

```
207.   private Class type;
208.   private String name;
209.   private Object value;
210.   private ArrayList<Field> fields;
211. }
```

API javax.swing.tree.TreeModel 1.2

- `Object getRoot()`

  Возвращает корневой узел.

- `int getChildCount(Object parent)`

  Возвращает количество дочерних узлов для заданного родительского узла.

- `Object getChild(Object parent, int index)`

  Возвращает дочерний узел заданного родительского узла, соответствующий указанному индексу.

- `int getIndexOfChild(Object parent, Object child)`

  Возвращает индекс указанного дочернего узла для заданного родительского узла или `-1`, если `child` не является дочерним узлом для `parent`.

- `boolean isLeaf(Object node)`

  Возвращает значение `true`, если указанный узел является листом дерева.

- `void addTreeModelListener(TreeModelListener l)`
- `void removeTreeModelListener(TreeModelListener l)`

  Создает или удаляет обработчик, который уведомляется о происходящих в дереве изменениях.

- `void valueForPathChanged(TreePath path, Object newValue)`

  Вызывается в тех случаях, когда редактор узлов изменяет значение узла.

Параметры:	`path`	Путь к редактируемому узлу
	`value`	Новое значение, возвращаемое редактором

API javax.swing.event.TreeModelListener 1.2

- `void treeNodesChanged(TreeModelEvent e)`
- `void treeNodesInserted(TreeModelEvent e)`
- `void treeNodesRemoved(TreeModelEvent e)`
- `void treeStructureChanged(TreeModelEvent e)`

  Вызывается моделью при изменении узлов дерева.

API javax.swing.event.TreeModelEvent 1.2

- `TreeModelEvent(Object eventSource, TreePath node)`

  Создает событие модели дерева.

Параметры:	`eventSource`	Модель дерева, которая генерирует данное событие
	`node`	Путь к узлу, подвергшемуся изменениям

Текстовые компоненты

На рис. 6.35 показаны все текстовые компоненты, включенные в библиотеку Swing. Три наиболее популярных компонента из этой библиотеки вы уже видели: это `JTextField`, `JPasswordField` и `JTextArea` (см. главу 9 первого тома). В следующих разделах мы расскажем об оставшихся текстовых компонентах. Мы обсудим также компонент `JSpinner`, содержащий поле форматированного текста и крошечные кнопки "up" и "down" для изменения их содержимого.

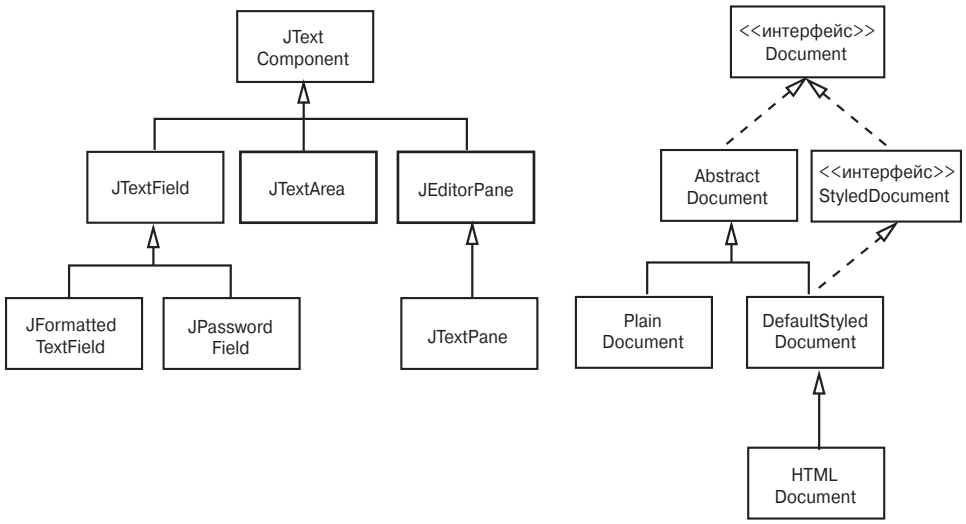


Рис. 6.35. Иерархическая структура текстовых компонентов и документов

Все текстовые компоненты отображают и редактируют данные, хранящиеся в объектной модели класса, который реализует интерфейс `Document`. Компоненты `JTextField` и `JTextArea` используют `PlainDocument`, который просто хранит последовательность строк простого текста без какого-либо форматирования.

`JEditorPane` может отображать и редактировать стилизованный текст (включающий шрифты, цвета и т.п.) в самых разных форматах, самым распространенным из которых является HTML; см. раздел "Отображение HTML-данных посредством панели `JEditorPane`" далее в этой главе. Интерфейс `StyledDocument` описывает дополнительные требования к стилям, шрифтам и цветам. Класс `HTMLDocument` реализует этот интерфейс.

Подкласс `JTextPane` класса `JEditorPane` также хранит стилизованный текст и внедряемые компоненты Swing. В этой книге мы не рассматриваем крайне сложный класс `JTextPane`, и вместо этого рекомендуем вам прочитать издание *Core Swing: Advanced Programming*, написанное Кимом Топли (Kim Topley). Для обычного использования класса `JTextPane` взгляните на пример `StylePad`, входящий в состав в JDK.

Изменение отслеживания в текстовых компонентах

Сложности при работе с интерфейсом `Document` возникают только тогда, когда вы пытаетесь реализовать свой собственный текстовый редактор. Тем не менее, существует наиболее распространенный случай использования этого интерфейса: отслеживание изменений.

Иногда бывает необходимо обновлять часть пользовательского интерфейса всякий раз, когда пользователь редактирует текст, не ожидая, пока пользователь щелкнет на кнопке. Ниже представлен простой пример. Мы отображаем три текстовых поля для красного, голубого и зеленого компонентов цвета. Всякий раз, когда изменяется содержимое текстовых полей, необходимо обновлять цвета. На рис. 6.36 показано работающее приложение, код которого представлен в листинге 6.12.



Рис. 6.36. Отслеживание изменений в текстовом поле

Прежде всего, обратите внимание, что отслеживание нажатий клавиш на клавиатуре — это не совсем подходящая идея. Дело в том, что некоторые нажатия клавиш (например, клавиш управления курсором) не приводят к изменению текста. Текст может быть обновлен в результате действий, производимых с помощью мыши (например, использование средней кнопки мыши для вставки данных в системе X11). Вместо этого, вам нужно "попросить" *документ* (а не компонент текста) уведомлять вас всякий раз, когда производится изменение данных. Для этого потребуется установить *слушатель документа* (document listener):

```
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 вставке или удалении символов. Третий метод для текстовых полей не вызывается вообще. Для более сложных типов документов он может быть вызван при некотором другом изменении, таком как изменение в форматировании. К сожалению, нет возможности сообщить об изменении текста посредством одиночного обратного вызова — как правило, вам не следует беспокоиться о том, как он изменяется. Более того, на этот случай даже нет специального класса. Таким образом, ваш слушатель документа должен реализовывать все три метода. Ниже показано, что предпринимается в программе:

```
DocumentListener listener = new DocumentListener()
      {
            public void insertUpdate(DocumentEvent event) { setColor(); }
            public void removeUpdate(DocumentEvent event) { setColor(); }
            public void changedUpdate(DocumentEvent event) {}
       }
```

Метод `setColor` использует метод `getText` для получения строк, вводимых пользователем в данный момент времени, из текстовых полей и установки цвета.

Наша программа имеет одно ограничение. Пользователь может ввести в текстовом поле текст с ошибками (например, "мешина" вместо "машина") или вообще оставить его пустым. Пока что мы перехватываем исключение `NumberFormatException`, генерируемое методом `parseInt`, и просто не обновляем цвет, если запись в текстовом поле не является числом. В следующем разделе будет показано, как можно предотвратить ввод пользователями неправильных данных.



Вместо того чтобы прослушивать события документа, можно добавить в поле слушатель события действия. Слушателю действия посылается уведомление всякий раз, когда пользователь нажимает клавишу <Enter>. Мы не рекомендуем применять такой подход, так как пользователи не всегда помнят о том, что нужно нажимать клавишу <Enter> после завершения ввода данных. Если вы используете слушатель действий, вам необходимо также установить слушатель фокуса, чтобы вы могли не пропустить момент, когда пользователь покинет текстовое поле.

Листинг 6.12. Содержимое файла `ChangeTrackingTest.java`

```
1.  import java.awt.*;
2.  import javax.swing.*;
3.  import javax.swing.event.*;
4.
5.  /**
6.   * @version 1.40 2007-08-05
7.   * @author Cay Horstmann
8.   */
9.
10. public class ChangeTrackingTest
11.
12. {
13.     public static void main(String[] args)
14.     {
15.         EventQueue.invokeLater(new Runnable()
16.             {
17.                 public void run()
18.                 {
19.                     ColorFrame frame = new ColorFrame();
20.                     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.                     frame.setVisible(true);
22.                 }
23.             });
24.     }
25. }
26.
27. /**
28.  * Фрейм с тремя текстовыми полями для установления фонового цвета.
29.  */
30. class ColorFrame extends JFrame
31. {
32.     public ColorFrame()
33.     {
34.         setTitle("ChangeTrackingTest");
35.
36.         DocumentListener listener = new DocumentListener()
37.             {
38.                 public void insertUpdate(DocumentEvent event)
39.                 {
40.                     setColor();
41.                 }
42.
43.                 public void removeUpdate(DocumentEvent event)
44.                 {
45.                     setColor();
46.                 }
```

```
47.                  public void changedUpdate(DocumentEvent event)
48.                  {
49.                  }
50.              };
51.
52.          panel = new JPanel();
53.
54.          panel.add(new JLabel("Red:"));
55.          redField = new JTextField("255", 3);
56.          panel.add(redField);
57.          redField.getDocument().addDocumentListener(listener);
58.
59.          panel.add(new JLabel("Green:"));
60.          greenField = new JTextField("255", 3);
61.          panel.add(greenField);
62.          greenField.getDocument().addDocumentListener(listener);
63.
64.          panel.add(new JLabel("Blue:"));
65.          blueField = new JTextField("255", 3);
66.          panel.add(blueField);
67.          blueField.getDocument().addDocumentListener(listener);
68.
69.          add(panel);
70.          pack();
71.      }
72.
73.      /**
74.       * Устанавливаем фоновый цвет для значений, хранящихся в текстовом поле.
75.       */
76.      public void setColor()
77.     {
78.          try
79.          {
80.              int red = Integer.parseInt(redField.getText().trim());
81.              int green = Integer.parseInt(greenField.getText().trim());
82.              int blue = Integer.parseInt(blueField.getText().trim());
83.              panel.setBackground(new Color(red, green, blue));
84.          }
85.          catch (NumberFormatException e)
86.          {
87.              // Не устанавливайте цвет, если введенные данные
88.              // невозможно проанализировать.
89.          }
90.      }
91.
92.      private JPanel panel;
93.      private JTextField redField;
94.      private JTextField greenField;
95.      private JTextField blueField;
96.  }
```

API javax.swing.JComponent 1.2

- `Dimension getPreferredSize()`
- `Void setPreferredSize(Dimension d)`

  Получает или устанавливает предпочтительный размер данного компонента.

API javax.swing.text.Document 1.2

- `int getLength()`
  Возвращает количество символов, которые содержит документ в данный момент времени.
- `String getText(int offset, int length)`
  Возвращает текст, содержащийся в данной части документа.

  Параметры: `offset` Начало текста
  `length` Длина желаемой строки
- `void addDocumentListener(DocumentListener listener)`
  Регистрирует слушатель на получение уведомлений в случае изменения документа.

API javax.swing.event.DocumentEvent 1.2

- `Document getDocument()`
  Получает документ, являющийся источником события.

API javax.swing.event.DocumentListener 1.2

- `void changedUpdate(DocumentEvent event)`
  Вызывается всякий раз, когда изменяется атрибут или набор атрибутов.
- `void insertUpdate(DocumentEvent event)`
  Вызывается всякий раз, когда происходит вставка текста в документ.
- `void removeUpdate(DocumentEvent event)`
  Вызывается всякий раз, когда удаляется часть документа.

Поля форматированного ввода

В последнем примере программы мы хотели, чтобы ее пользователь вводил числа, а не произвольные строки. То есть пользователю разрешается вводить только цифры от `0` до `9` и дефис (`-`). Дефис, если он будет присутствовать во введенных данных, должен быть первым символом строки ввода.

На первый взгляд, проверка такого ввода не является сложной задачей. Мы можем установить слушатель клавиш в текстовом поле, а затем удалять все события клавиш, которые не связаны с вводом цифр или дефиса. К сожалению, этот простой подход, хоть он обычно рекомендуется в качестве метода проверки вводимых данных, на практике работает со сбоями. Во-первых, не всякая комбинация допустимых символов ввода является допустимым числом. Например, `--3` и `3-3` не являются допустимыми комбинациями, несмотря даже на то, что они состоят из допустимых символов. Но самое главное то, что есть другие способы изменения текста, не включающего ввод символов с помощью клавиш. В зависимости от предпочтений пользователя, для вырезания, копирования и вставки текста могут использоваться некоторые комбинации (сочетания) клавиш. Например, с помощью комбинации <Ctrl+V> производится вставка содержимого из буфера в текстовое поле. То есть нам тоже нужно наблюдать за тем, чтобы пользователь не вставил недействительный символ. Понятно, что попытка отфильтровывать нажатия клавиш, гарантирующая, что содержимое текстового поля всегда будет действительным, начинает казаться настоящей рутиной. Как правило, для программиста такие задачи уже должны быть решены.

Может быть, это покажется удивительным, но до выхода Java SE 1.4 не существовало компонентов для ввода числовых значений. Начиная с первой версии Core Java, мы предоставляем реализацию для `IntTextField` — текстового поля для ввода целого числа с подходящим форматированием. В каждой новой версии Core Java мы изменяли реализацию, чтобы воспользоваться любым преимуществом от полуготовых схем проверки, добавляемых в каждую версию Java. Наконец, в Java SE 1.4, разработчики Swing создали класс `JFormattedTextField`, который можно использовать не только для ввода чисел, но и для ввода дат и более экзотических значений, таких как IP-адреса.

Ввод целых чисел

Давайте начнем с простого случая: текстовое поле для ввода целых чисел:

```
JFormattedTextField intField =
   new JFormattedTextField(NumberFormat.getIntegerInstance());
```

Метод `NumberFormat.getIntegerInstance()` возвращает объект, который форматирует целые числа на основе текущей местной специфики. В США, например, запятые используются в качестве десятичных разделителей, позволяя пользователям вводить значения вроде `1,729`. В главе 5 мы рассказывали о том, как можно выбрать местную специфику.

Как и для любого другого текстового поля, вы можете установить несколько столбцов:

```
intField.setColumns(6);
```

Вы можете задать значение по умолчанию с помощью метода `setValue`. Этот метод принимает параметр `Object`, поэтому вам необходимо представить значение `int` как объект `Integer`:

```
intField.setValue(new Integer(100));
```

Как правило, пользователи вводят данные в нескольких текстовых полях, а затем щелкают на кнопке, чтобы прочитать все значения. При щелчке на кнопке вы можете получить значение, введенное пользователем, с помощью метода `getValue`. Этот метод возвращает результат `Object`, поэтому вам потребуется привести его к соответствующему типу. `JFormattedTextField` возвращает объект, имеющий тип `Long`, если пользователь отредактировал значение. А если пользователь не делал никаких изменений, возвращается исходный объект `Integer`. Таким образом, вам необходимо привести возвращаемое значение к обычному суперклассу `Number`:

```
Number value = (Number) intField.getValue();
int v = value.intValue();
```

Поле форматированного текста не представляет особого интереса до тех пор, пока вы не проанализируете, что произойдет, если пользователь введет недействительные данные. Этому вопросу посвящен следующий раздел.

Поведение при потере фокуса

Давайте посмотрим на то, что происходит, когда пользователь вводит данные в текстовом поле. Пользователь вводит что-то, и в один момент решает покинуть это поле, возможно, посредством щелчка кнопкой мыши на другом компоненте. В этом случае текстовое поле *теряет фокус*. Мигающий курсор больше не появляется в текстовом поле, а нажатия клавиш клавиатуры направляются в другой компонент.

Когда поле форматированного текста теряет фокус, объект форматирования "смотрит" на строку текста, введенную пользователем. Если объект знает о том, как производится преобразование текстовой строки в объект, значит, текст является действительным. В противном случае он не является таковым. Чтобы проверить, являет-

ся ли текущее содержимое текстового поля действительным, вы можете использовать метод `isEditValid`.

Поведение по умолчанию при потере фокуса называется "принять или вернуть". Если текстовая строка является действительной, она принимается. Объект форматирования преобразовывает ее в объект. Этот объект становится текущим значением поля (то есть, возвращаемое значение метода `getValue`, с которым вы уже сталкивались в предыдущем разделе). Затем значение преобразовывается обратно в строку, которая становится текстовой строкой, видимой в поле. Например, объект форматирования целых чисел распознает введенную комбинацию `1729` как действительную, устанавливает текущее значение как `new Long(1729)`, а затем преобразовывает его обратно в строку, используя запятую в качестве десятичного разделителя: `1,729`.

И наоборот, если текстовая строка является недействительной, текущее значение не изменяется, и текстовое поле возвращается в строку, представляющую старое значение. Например, если пользователь вводит неправильное значение, такое как `x1`, то при потере фокуса будет восстановлено старое значение.



Объект форматирования целых чисел считает текстовую строку действительной, если она начинается с целого числа. Например, `1729x` является действительной строкой. Она преобразовывается в число `1729`, которое затем преобразовывается в строку `1,729`.

С помощью метода `setFocusLostBehavior` вы можете установить другие варианты поведения. Поведение "принять" немногим отличается от поведения по умолчанию. Если текстовая строка является недействительной, то текстовая строка и значение поля остаются неизмененными — с этого момента они не являются синхронизированными. Поведение "продолжение" (persist) является еще более консервативным. Даже если текстовая строка является действительной, ни текстовое поле, ни текущее значение не будут изменяться. Вы можете даже вызвать метод `commitEdit` или `setText`, чтобы снова синхронизировать их. Наконец, есть еще поведение "отклонить", которое вообще кажется малополезным. Всякий раз при потере фокуса введенные пользователем данные игнорируются, и текстовая строка принимает старое значение.



Обычно по умолчанию используется именно поведение "принять или отклонить". Тем не менее, следует предостеречь об одной возможной проблеме. Предположим, что диалоговое окно содержит текстовое поле для целочисленного значения. Пользователь вводит `" 1729"` с пробелом в качестве первого символа и щелкает на кнопке OK. Этот пробел делает число недействительным, и значение поля возвращается к старому значению. Слушатель события кнопки OK получает значение поля и закрывает диалоговое окно. Пользователь никогда не узнает, что новое значение было отклонено. В этой ситуации выбрать поведение "принять" и заставить слушателя кнопки OK проверять, чтобы все правки в поле были действительными, прежде чем закрывать диалоговое окно.

Фильтры

Базовая функциональность полей форматированного текста является простой и достаточной для большинства случаев. Тем не менее, вы можете добавить пару улучшений. Возможно, вы хотите сделать так, чтобы пользователь вообще не смог ввести нецифровые значения. Это можно сделать посредством *фильтра документа* (document filter). Вспомните, что в архитектуре "модель–представление–контроллер"

последний преобразовывает входящие события в команды и изменяет базовый документ текстового поля, другими словами — текстовую строку, хранящуюся в объекте PlainDocument. Например, всякий раз, когда контроллер обрабатывает команду вставки текста в документ, он вызывает команду "вставить строку". Строка, которую необходимо вставить, может быть представлена или одним символом, или содержимым буфера вставки. Фильтр документа может перехватить эту команду и изменить строку или отменить вставку вообще. Ниже показан код метода insertString фильтра, анализирующего вставляемую строку и вставляющего только те символы, которые являются цифрами или знаком -. (Код обрабатывает вспомогательные символы кодировки Unicode, о чем было рассказано в главе 3. В главе 12 вы найдете описание класса StringBuilder.)

```
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);
}
```

Вам нужно также переопределить метод replace класса DocumentFilter — он вызывается при выборе текста и его последующей вставке. Реализация метода replace не представляет сложности — см. код, представленный в листинге 6.13.

Теперь необходимо установить фильтр документа. К сожалению, не существует простого способа сделать это. Вам необходимо переопределить метод getDocumentFilter класса объекта форматирования, и передать объект этого класса в JFormattedTextField.

Целочисленное текстовое поле использует InternationalFormatter, который инициализируется посредством метода NumberFormat.getIntegerInstance(). Ниже показано, как производится установка объекта форматирования для получения желаемого фильтра:

```
JFormattedTextField intField = new JFormattedTextField(new
        InternationalFormatter(NumberFormat.getIntegerInstance())
        {
            protected DocumentFilter getDocumentFilter()
            {
                return filter;
            }
            private DocumentFilter filter = new IntFilter();
        });
```



В документации по Java SE сказано, что класс `DocumentFilter` был разработан для того, чтобы избавиться от создания подклассов. До появления версии Java SE 1.3 фильтрация в текстовом поле достигалась путем расширения класса `PlainDocument` и переопределением методов `insertString` и `replace`. Теперь класс `PlainDocument` имеет подключаемый фильтр. Это очень хорошее изменение. Оно могло бы быть еще лучше, если бы фильтр был сделан подключаемым в классе объекта форматирования. Увы, этого не сделано, и мы должны создавать подкласс объекта форматирования.

Попытайтесь выполнить пример программы `FormatTest`, код которой представлен в конце этого раздела. Для третьего текстового поля был установлен фильтр. Вы можете вставить только цифры или знак минус (`-`). Обратите внимание, что вы можете по-прежнему вводить недействительные строки, такие как `"1-2-3"`. Вообще говоря, невозможно защититься от всех недействительных строк посредством фильтрации. Например, строка `"-"` является недействительной, хотя фильтр не может отклонить ее, поскольку это приставка действительной строки `"-1"`. Несмотря даже на то, что фильтры не могут обеспечить идеальную защиту, имеет смысл использовать их для отклонения входных данных, которые явным образом являются недействительными.



Другой вариант использования фильтрации заключается в том, чтобы перевести все символы в верхний регистр. Написать код для такого фильтра несложно. В методах `insertString` и `replace` фильтра необходимо перевести вставляемую строку в верхний регистр, а затем вызвать метод подкласса.

Средства проверки

Существует еще один потенциально полезный механизм уведомления пользователей о неверно введенных данных. Вы можете присоединить *средство проверки* (verifier) к любому компоненту класса `JComponent`. Если компонент теряет фокус, производится запрос на выполнение средства проверки. Если средство проверки сообщает, что содержимое компонента является недействительным, компонент тут же вновь получает фокус. Таким образом, пользователю придется устранить ошибку во введенных данных, прежде чем вводить другие данные.

Средство проверки должно расширять абстрактный класс `InputVerifier` и определять метод `verify`. Определить средство проверки для полей форматированного текста несложно. Метод `isEditValid` класса `JFormattedTextField` вызывает объект форматирования и возвращает значение `true`, если объект форматирования преобразует текстовую строку в объект. Ниже представлен код средства проверки:

```
class FormattedTextFieldVerifier extends InputVerifier
{
    public boolean verify(JComponent component)
    {
        JFormattedTextField field = (JFormattedTextField) component;
        return field.isEditValid();
    }
}
```

Вы можете присоединить его к любому `JFormattedTextField`:

```
intField.setInputVerifier(new FormattedTextFieldVerifier());
```

Однако и средство проверки не является исключительно надежным. Если щелкнуть на кнопке, кнопка уведомит слушателей событий до того, как недействительный

компонент вновь получит фокус. После этого слушатели событий могут получить недействительный результат от компонента, который не прошел проверку. Такое поведение объясняется следующим: пользователи могут захотеть щелкнуть на кнопке Cancel (Отмена), не устранив ошибку во введенных данных.

Четвертое текстовое поле в примере программы имеет средство проверки. Попытайтесь ввести недействительное число (например, `x1729`) и нажмите клавишу <Tab> или щелкните кнопкой мыши на другом текстовом поле. Обратите внимание, что поле тут же получит фокус. Однако если вы щелкнете на кнопке OK, слушатель события вызовет метод `getValue`, который сообщит о последнем действительном значении.

Другие стандартные средства форматирования

Помимо целочисленных, `JFormattedTextField` поддерживает и ряд других форматировщиков. Класс `NumberFormat` имеет следующие статические методы:

```
getNumberInstance
getCurrencyInstance
getPercentInstance
```

Эти методы создают форматировщики чисел с плавающей запятой, валют и процентов. Например, чтобы получить текстовое поле для ввода денежных сумм, нужно вызвать метод

```
JFormattedTextField currencyField =
   new JFormattedTextField(NumberFormat.getCurrencyInstance());
```

Чтобы отредактировать даты и времена, нужно вызвать один из статических методов класса `DateFormat`:

```
getDateInstance
getTimeInstance
getDateTimeInstance
```

Например:

```
JFormattedTextField dateField =
   new JFormattedTextField(DateFormat.getDateInstance());
```

В этом поле редактируется дата в формате, принятом по умолчанию:

```
Aug 5, 2007
```

Вместо него можно выбрать другой формат, с более короткой записью:

```
8/5/07
```

Для этого потребуется вызвать следующий метод:

```
DateFormat.getDateInstance(DateFormat.SHORT)
```



По умолчанию форматом даты является так называемый “снисходительный” формат. Другими словами, несуществующая дата вроде February 31, 2002 будет преобразована в March 3, 2002. Такое поведение может вызвать удивление у пользователей. В этом случае необходимо вызвать метод `setLenient(false)` в объекте `DateFormat`.

`DefaultFormatter` может форматировать объекты любого класса, который имеет конструктор со строковым параметром и соответствующий метод `toString`. Например, класс URL имеет конструктор `URL(String)`, который можно использовать для создания URL из строки, например:

```
URL url = new URL("http://java.sun.com");
```

Таким образом, вы можете использовать DefaultFormatter для форматирования объектов URL. Форматировщик вызывает метод toString в значении поля, чтобы инициализировать текстовое поле. Когда поле теряет фокус, форматировщик создает новый объект такого же класса, что и класс текущего значения, с помощью конструктора с параметром String. Если этот конструктор генерирует исключение, то редактирование не является действительным. Вы можете испытать это в примере программы, введя URL-адрес без приставки "http:".



По умолчанию DefaultFormatter работает в *режиме перезаписи* (overwrite mode). В этом заключается его отличие от остальных форматировщиков. Данный режим не является полезным, поэтому его можно отключить посредством вызова метода setOverwriteMode(false).

Наконец, MaskFormatter полезен для шаблонов фиксированного размера, содержащих некоторую константу и некоторые переменные символы. Например, номера карточек социального страхования в США (такие как 078-05-1120) могут принять следующий формат:

```
new MaskFormatter("###-##-####")
```

Символ # обозначает одну цифру. В табл. 6.3 показаны символы, которые вы можете использовать в форматировщике маски.

Таблица 6.3. Символы MaskFormatter

Символ	Описание
#	Цифра
?	Буква
U	Буква, переведенная в верхний регистр
L	Буква, переведенная в нижний регистр
A	Буква или цифра
H	Шестнадцатеричная цифра [0-9A-Fa-f]
*	Любой символ
'	Символ отмены, который позволяет включить символ в шаблон

Вы можете ограничить символы, которые можно вводить в поле, посредством вызова одного из методов класса MaskFormatter:

```
setValidCharacters
setInvalidCharacters
```

Например, чтобы прочитать буквенную оценку академической успеваемости (например, A+ или F), можно использовать следующий метод:

```
MaskFormatter formatter = new MaskFormatter("U*");
formatter.setValidCharacters("ABCDF+- ");
```

Однако определить, что второй символ не может быть буквой, нельзя.

Обратите внимание, что строка, которая форматируется форматировщиком маски, имеет точно такую же длину, что и маска. Если пользователь удаляет символы во время редактирования, они будут заменяться *символом-заполнителем* (placeholder character). По умолчанию в качестве этого символа используется пробел, однако вы можете заменить его с помощью метода setPlaceholderCharacter.

Например:

```
formatter.setPlaceholderCharacter('0');
```

По умолчанию форматировщик маски работает в режиме замены вводимых данных, понять который вы сможете интуитивно, если попробуете его в представленном нами примере. Также обратите внимание, что позиция курсора "скачет" по фиксированным символам в маске.

Форматировщики масок очень эффективны в случае использования строгих шаблонов, таких как номера пенсионных удостоверений в США или номера телефонов в США. Необходимо учитывать то, что в шаблонах маски не допускается применять переменные значения, поэтому, например, вы не сможете использовать форматировщик маски для международных номеров телефонов, имеющих разное количество цифр.

Специальные форматировщики

Если ни один из стандартных форматировщиков не подходит, можно без труда определить собственный. Возьмем, к примеру, 4-байтные IP-адреса:

```
130.65.86.66
```

Вы не можете использовать `MaskFormatter`, поскольку каждый байт может быть представлен одной, двумя или тремя цифрами. Также в форматировщике необходимо проверить, что значение в каждом байте не превышает 255.

Чтобы определить собственный форматировщик, необходимо расширить класс `DefaultFormatter` и переопределить методы:

```
String valueToString(Object value)
Object stringToValue(String text)
```

Первый метод преобразовывает значение поля в строку, которая отображается в текстовом поле. Второй метод анализирует введенный пользователем текст и преобразовывает его обратно в объект. Если один из этих методов обнаружит ошибку, он должен сгенерировать исключение `ParseException`.

В нашей программе мы храним IP-адреса в массиве `byte[]`, длина которого составляет `4`. Метод `valueToString` формирует строку, разделяющую байты точками. Обратите внимание, что значения `byte` могут принимать значения 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 беззнаковые целочисленные значения, необходимо прибавить 256.

```
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();
}
```

И наоборот, метод `stringToValue` анализирует строку и создает объект `byte[]`, если строка является действительной. Если нет, генерируется исключение `ParseException`.

```
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;
}
```

Попытайтесь ввести данные в поле IP-адреса в нашей программе. Если вы введете недействительный адрес, в поле будет отображен последний действительный адрес.

Программа, код которой представлен в листинге 6.13, показывает разные поля форматированного текста в действии (рис. 6.37). Щелкните на кнопке OK, чтобы получить текущие значения из полей.



По адресу `http://java.sun.com/products/jfc/tsc/articles/reftf/` хранится краткая статья, в которой описывается форматировщик, совпадающий с регулярным выражением.

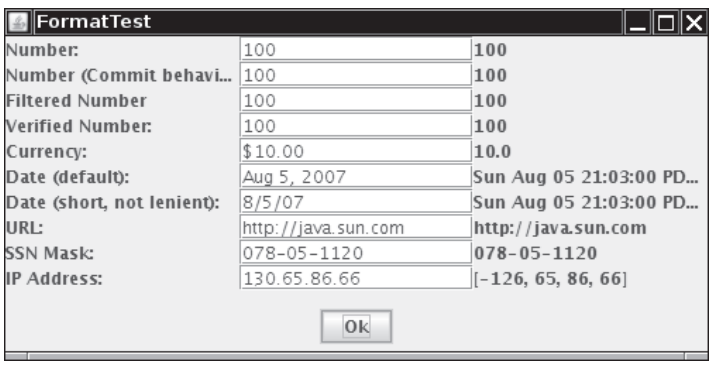


Рис. 6.37. Программа `FormatTest`

Листинг 6.13. Содержимое файла `FormatTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.net.*;
4.  import java.text.*;
5.  import java.util.*;
6.  import javax.swing.*;
7.  import javax.swing.text.*;
8.
9.  /**
10.  * Программа для проверки полей форматированного текста.
11.  * @version 1.02 2007-06-12
12.  * @author Cay Horstmann
13.  */
14. public class FormatTest
15. {
16.    public static void main(String[] args)
17.    {
18.       EventQueue.invokeLater(new Runnable()
19.       {
20.          public void run()
21.          {
22.             FormatTestFrame frame = new FormatTestFrame();
23.             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24.             frame.setVisible(true);
25.          }
26.       });
27.    }
28. }
29.
30. /**
31.  * Фрейм с набором полей форматированного текста
32.  * и кнопка, отображающая значения полей.
33.  */
34. class FormatTestFrame extends JFrame
35. {
36.    public FormatTestFrame()
37.    {
38.       setTitle("FormatTest");
39.       setSize(WIDTH, HEIGHT);
40.
41.       JPanel buttonPanel = new JPanel();
42.       okButton = new JButton("Ok");
43.       buttonPanel.add(okButton);
44.       add(buttonPanel, BorderLayout.SOUTH);
45.
46.       mainPanel = new JPanel();
47.       mainPanel.setLayout(new GridLayout(0, 3));
48.       add(mainPanel, BorderLayout.CENTER);
49.
50.       JFormattedTextField intField =
51.       new JFormattedTextField(NumberFormat.getIntegerInstance());
52.       intField.setValue(new Integer(100));
53.       addRow("Number:", intField);
54.
55.       JFormattedTextField intField2 =
```

```
56.         new JFormattedTextField(NumberFormat.getIntegerInstance());
57.       intField2.setValue(new Integer(100));
58.       intField2.setFocusLostBehavior(JFormattedTextField.COMMIT);
59.       addRow("Number (Commit behavior):", intField2);
60.
61.       JFormattedTextField intField3 = new JFormattedTextField(
62.            new InternationalFormatter( NumberFormat.getIntegerInstance())
63.          {
64.             protected DocumentFilter getDocumentFilter()
65.             {
66.                return filter;
67.             }
68.
69.             private DocumentFilter filter = new IntFilter();
70.          });
71.       intField3.setValue(new Integer(100));
72.       addRow("Filtered Number", intField3);
73.
74.       JFormattedTextField intField4 =
75.       new JFormattedTextField(NumberFormat.getIntegerInstance());
76.       intField4.setValue(new Integer(100));
77.       intField4.setInputVerifier(new FormattedTextFieldVerifier());
78.       addRow("Verified Number:", intField4);
79.
80.       JFormattedTextField currencyField =
81.          new JFormattedTextField(NumberFormat.getCurrencyInstance());
82.       currencyField.setValue(new Double(10));
83.       addRow("Currency:", currencyField);
84.
85.       JFormattedTextField dateField =
86.          new JFormattedTextField(DateFormat.getDateInstance());
87.       dateField.setValue(new Date());
88.       addRow("Date (default):", dateField);
89.
90.       DateFormat format = DateFormat.getDateInstance(DateFormat.SHORT);
91.       format.setLenient(false);
92.       JFormattedTextField dateField2 = new JFormattedTextField(format);
93.       dateField2.setValue(new Date());
94.       addRow("Date (short, not lenient):", dateField2);
95.
96.       try
97.       {
98.          DefaultFormatter formatter = new DefaultFormatter();
99.          formatter.setOverwriteMode(false);
100.         JFormattedTextField urlField = new JFormattedTextField(formatter);
101.         urlField.setValue(new URL("http://java.sun.com"));
102.         addRow("URL:", urlField);
103.      }
104.      catch (MalformedURLException e)
105.      {
106.         e.printStackTrace();
107.      }
108.
109.      try
110.      {
111.         MaskFormatter formatter = new MaskFormatter("###-##-####");
112.         formatter.setPlaceholderCharacter('0');
```

```
113.        JFormattedTextField ssnField = new JFormattedTextField(formatter);
114.        ssnField.setValue("078-05-1120");
115.        addRow("SSN Mask:", ssnField);
116.     }
117.     catch (ParseException exception)
118.     {
119.        exception.printStackTrace();
120.     }
121.
122.     JFormattedTextField ipField = new JFormattedTextField(new IPAddressFormatter());
123.     ipField.setValue(new byte[] { (byte) 130, 65, 86, 66 });
124.     addRow("IP Address:", ipField);
125.  }
126.
127.  /**
128.   * Добавляет строку в главную панель.
129.   * @param labelText метка поля
130.   * @param field образец поля
131.   */
132.  public void addRow(String labelText, final JFormattedTextField field)
133.  {
134.     mainPanel.add(new JLabel(labelText));
135.     mainPanel.add(field);
136.     final JLabel valueLabel = new JLabel();
137.     mainPanel.add(valueLabel);
138.     okButton.addActionListener(new ActionListener()
139.        {
140.           public void actionPerformed(ActionEvent event)
141.           {
142.              Object value = field.getValue();
143.              Class<?> cl = value.getClass();
144.              String text = null;
145.              if (cl.isArray())
146.              {
147.                 if (cl.getComponentType().isPrimitive())
148.                 {
149.                    try
150.                 {
151.                    text = Arrays.class.getMethod("toString",
152.                       cl).invoke(null, value).toString();
153.                 }
154.                 catch (Exception ex)
155.                 {
156.                    // Игнорируем исключения рефлексии.
157.                 }
158.              }
159.              else text = Arrays.toString((Object[]) value);
160.           }
161.           else text = value.toString();
162.           valueLabel.setText(text);
163.        }
164.     });
165.  }
166.
167.  public static final int WIDTH = 500;
168.  public static final int HEIGHT = 250;
169.
```

```
170.    private JButton okButton;
171.    private JPanel mainPanel;
172. }
173.
174. /**
175.  * Фильтр, ограничивающий ввод цифр и знака '-'.
176.  */
177. class IntFilter extends DocumentFilter
178. {
179.    public void insertString(FilterBypass fb, int offset, String string,
180.                             AttributeSet attr)
181.       throws BadLocationException
182.    {
183.       StringBuilder builder = new StringBuilder(string);
184.       for (int i = builder.length() - 1; i >= 0; i--)
185.       {
186.          int cp = builder.codePointAt(i);
187.          if (!Character.isDigit(cp) && cp != '-')
188.          {
189.             builder.deleteCharAt(i);
190.             if (Character.isSupplementaryCodePoint(cp))
191.             {
192.                i--;
193.                builder.deleteCharAt(i);
194.             }
195.          }
196.       }
197.       super.insertString(fb, offset, builder.toString(), attr);
198.    }
199.
200.    public void replace(FilterBypass fb, int offset, int length, String string,
201.                        AttributeSet attr)
202.       throws BadLocationException
203.    {
204.       if (string != null)
205.       {
206.          StringBuilder builder = new StringBuilder(string);
207.          for (int i = builder.length() - 1; i >= 0; i--)
208.          {
209.             int cp = builder.codePointAt(i);
210.             if (!Character.isDigit(cp) && cp != '-')
211.             {
212.                builder.deleteCharAt(i);
213.                if (Character.isSupplementaryCodePoint(cp))
214.                {
215.                   i--;
216.                   builder.deleteCharAt(i);
217.                }
218.             }
219.          }
220.          string = builder.toString();
221.       }
222.       super.replace(fb, offset, length, string, attr);
223.    }
224. }
225.
```

```
226. /**
227.  * Средство проверки действительности поля форматированного текста.
228.  */
229. class FormattedTextFieldVerifier extends InputVerifier
230. {
231.    public boolean verify(JComponent component)
232.    {
233.    JFormattedTextField field = (JFormattedTextField) component;
234.    return field.isEditValid();
235.    }
236. }
237.
238. /**
239.  * Форматировщик для 4-байтных IP-адресов в форме a.b.c.d
240.  */
241. class IPAddressFormatter extends DefaultFormatter
242. {
243.    public String valueToString(Object value) throws ParseException
244.    {
245.       if (!(value instanceof byte[])) throw new ParseException("Not a byte[]", 0);
246.       byte[] a = (byte[]) value;
247.       if (a.length != 4) throw new ParseException("Length != 4", 0);
248.       StringBuilder builder = new StringBuilder();
249.       for (int i = 0; i < 4; i++)
250.       {
251.          int b = a[i];
252.          if (b < 0) b += 256;
253.          builder.append(String.valueOf(b));
254.          if (i < 3) builder.append('.');
255.       }
256.       return builder.toString();
257.    }
258.
259.    public Object stringToValue(String text) throws ParseException
260.    {
261.       StringTokenizer tokenizer = new StringTokenizer(text, ".");
262.       byte[] a = new byte[4];
263.       for (int i = 0; i < 4; i++)
264.       {
265.          int b = 0;
266.          if (!tokenizer.hasMoreTokens()) throw new ParseException("Слишком мало байт", 0);
267.          try
268.          {
269.             b = Integer.parseInt(tokenizer.nextToken());
270.          }
271.          catch (NumberFormatException e)
272.          {
273.             throw new ParseException("Not an integer", 0);
274.          }
275.          if (b < 0 || b >= 256) throw new ParseException("Byte out of range", 0);
276.          a[i] = (byte) b;
277.       }
278.       if (tokenizer.hasMoreTokens()) throw new ParseException("Too many bytes", 0);
279.       return a;
280.    }
281. }
```

API 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 указанный форматировщик. Обратите внимание, что классы `DefaultFormatter` и `InternationalFormatter` являются наследниками класса `JFormattedTextField.AbstractFormatter`.

- `Object getValue()`

  Возвращает текущее допустимое значение поля. Обратите внимание, что оно может не соответствовать редактируемой строке.

- `void setValue(Object value)`

  Пытается установить значение для данного объекта. Попытка завершится неудачей, если форматировщик не сумеет преобразовать объект в строку.

- `void commitEdit()`

  Пытается установить действительное значение поля из редактируемой строки. Попытка может завершиться неудачей, если форматировщик не сумеет преобразовать строку.

- `boolean isEditValid()`

  Проверяет, представляет ли редактируемая строка допустимое значение.

- `int getFocusLostBehavior()`
- `void setFocusLostBehavior(int behavior)`

  Получает или устанавливает поведение "потеря фокуса". `behavior` может принимать такие константы: `COMMIT_OR_REVERT`, `REVERT`, `COMMIT` и `PERSIST` класса `JFormattedTextField`.

API javax.swing.JFormattedTextField.AbstractFormatter 1.4

- `abstract String valueToString(Object value)`

  Преобразует значение в редактируемую строку. Генерирует исключение `ParseException`, если значение не является подходящим для данного форматировщика.

- `abstract Object stringToValue(String s)`

  Преобразует строку в значение. Генерирует исключение `ParseException`, если `s` не имеет подходящий формат.

- `DocumentFilter getDocumentFilter()`

  Переопределите этот метод, чтобы создать фильтр документа, ограничивающий ввод данных в текстовое поле. Возвращенное значение `null` свидетельствует о том, что фильтрация не нужна.

API javax.swing.text.DefaultFormatter 1.3

- `boolean getOverwriteMode()`
- `void setOverwriteMode(boolean mode)`

  Получает или устанавливает режим перезаписи. Если `mode` имеет значение `true`, новые символы перезапишут существующие символы при редактировании текста.

API javax.swing.text.DocumentFilter 1.4

- `void insertString(DocumentFilter.FilterBypass bypass, int offset, String text, AttributeSet attrib)`

  Вызывается перед вставкой строки в документ. Вы можете переопределить метод и модифицировать строку. Вы можете отключить вставку, не вызывая метод `super.insertString` или вызывая метод `bypass`, чтобы определить документ без фильтрации.

Параметры:		
	`bypass`	Объект, позволяющий выполнять команды, минующие фильтр
	`offset`	Смещение, на котором начнется вставка текста
	`text`	Символы для вставки
	`attrib`	Атрибуты форматирования вставляемого текста

- `void replace(DocumentFilter.FilterBypass bypass, int offset, int length, String text, AttributeSet attrib)`

  Вызывается перед заменой новой строки в определенной части документа. Вы можете переопределить этот метод и модифицировать строку. Вы можете отключить замену, не вызывая метод `super.replace` или вызывая метод `bypass`, чтобы изменить документ без фильтрации.

- `void remove(DocumentFilter.FilterBypass bypass, int offset, int length)`

  Вызывается перед удалением части документа. Если необходимо проанализировать удаление, документ можно получить посредством вызова метода `bypass.getDocument()`.

Параметры:		
	`bypass`	Объект, позволяющий выполнять команды редактирования, минующие фильтр
	`offset`	Смещение, на котором начнется вставка текста
	`length`	Длина заменяемой части

API javax.swing.text.MaskFormatter 1.4

- `MaskFormatter(String mask)`

  Создает форматировщик маски с данной маской. Символы, применяемые в маске, перечислены в табл. 6.3.

- `String getValidCharacters()`
- `void setValidCharacters(String characters)`

  Получает или устанавливает допустимые символы редактирования. Для переменных частей маски допускается использовать только символы в данной строке.

- `String getInvalidCharacters()`
- `void setInvalidCharacters(String characters)`

  Получает или устанавливает недопустимые символы редактирования.

- `char getPlaceholderCharacter()`
- `void setPlaceholderCharacter(char ch)`

  Получает или устанавливает символ-заполнитель, используемый для переменных символов в маске, которые пользователь еще не ввел. По умолчанию в качестве этого символа используется пробел.

- `String getPlaceholder()`
- `void setPlaceholder(String s)`

  Получает или устанавливает строку-заполнитель. "Хвост" этой строки используется, когда пользователь не ввел все переменные символы в маску. Если он равен `null` или если его длина короче маски, для заполнения маски будет использоваться символ-заполнитель.

- `boolean getValueContainsLiteralCharacters()`
- `void setValueContainsLiteralCharacters(boolean b)`

  Получает или устанавливает флаг "значение содержит буквенные символы". Если этот флаг равен `true`, значение поля содержит буквенные (непеременные) части маски. Если он равен `false`, буквенные символы удаляются. По умолчанию флаг имеет значение `true`.

Компонент `JSpinner`

Компонент `JSpinner` (циклический счетчик) содержит текстовое поле и две небольших кнопки. При щелчке на этих кнопках значение текстового поля увеличивается или уменьшается на заданную величину (рис. 6.38).

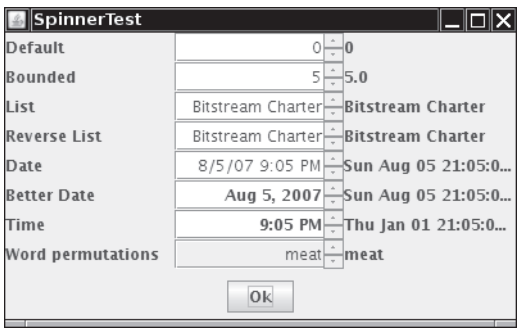


Рис. 6.38. Несколько вариантов исполнения компонента `JSpinner`

В качестве значений в счетчике могут использоваться числа, даты, значения из списка или, как это часто бывает, любая последовательность значений, для которых можно определить предшествующие и последующие значения.
По умолчанию счетчик работает с целыми числами, и кнопки увеличивают или уменьшают целое число на 1. Текущее значение в счетчике можно получить посредством вызова метода `getValue`. Этот метод возвращает объект `Object`. Его необходимо привести к типу `Integer` и получить значение:

```
JSpinner defaultSpinner = new JSpinner();
. . .
int value = (Integer) defaultSpinner.getValue();
```

Вы можете изменить величину приращения и задать нижние и верхние границы приращения. Ниже показан пример счетчика со стартовым значением `5`, ограниченным значениями `0` и `10`, и с приращением `0.5`:

```
JSpinner boundedSpinner = new JSpinner(new SpinnerNumberModel(5, 0, 10, 0.5));
```

Существует два конструктора `SpinnerNumberModel`, причем один из них имеет только параметры типа `int`, а другой — только параметры типа `double`. Если любой из этих параметров имеет тип числа с плавающей запятой, будет использоваться второй конструктор. Он выберет для значения счетчика объект `Double`.

Счетчики не ограничены числовыми значениями. Вы можете создать счетчик, который будет осуществлять перебор значений из любой коллекции значений. Просто передайте `SpinnerListModel` конструктору `JSpinner`. Вы можете создавать `SpinnerListModel` на основе любого массива или класса, реализующего интерфейс `List` (например, `ArrayList`). В нашей программе мы отображаем счетчик со всеми доступными именами шрифтов.

```
String[] fonts =
  GraphicsEnvironment.getLocalGraphicsEnvironment().getAvailableFontFamilyNames();
JSpinner listSpinner = new JSpinner(new SpinnerListModel(fonts));
```

Однако мы обнаружили, что направление итерации было несколько запутанным, поскольку оно отличалось от привычного пользователям направления выбора значений в комбинированных окнах. В комбинированном окне более высокие значения находились *под* более низкими значениями, поэтому можно было бы ожидать, что при нажатии кнопки со стрелкой вниз будет выбрано более высокое значение, однако в счетчиках все происходит наоборот. Изменить направление в `SpinnerListModel` невозможно, однако желаемый результат можно получить благодаря анонимному подклассу:

```
JSpinner reverseListSpinner = new JSpinner(
   new SpinnerListModel(fonts)
   {
      public Object getNextValue()
      {
         return super.getPreviousValue();
      }
      public Object getPreviousValue()
      {
         return super.getNextValue();
      }
   });
```

Посмотрите, как работают обе версии и вы поймете, какая из них является более интуитивной.

Другим хорошим вариантом прмиенения счетчика является выбор дат. Создать такой счетчик с текущей датой в качестве начальной можно следующим образом:

```
JSpinner dateSpinner = new JSpinner(new SpinnerDateModel());
```

Тем не менее, если посмотреть внимательно на рис. 6.38, вы увидите, что текст в счетчике отображает как дату, так и время:

```
8/05/07 7:23 PM
```

При выборе даты время не так уж важно. А чтобы счетчик показывал только дату, необходим следующий код:

```
JSpinner betterDateSpinner = new JSpinner(new SpinnerDateModel());
String pattern = ((SimpleDateFormat) DateFormat.getDateInstance()).toPattern();
betterDateSpinner.setEditor(new JSpinner.DateEditor(betterDateSpinner, pattern));
```

Используя такой же подход, можно сделать так, чтобы выбиралось только время:

```
JSpinner timeSpinner = new JSpinner(new SpinnerDateModel());
pattern = ((SimpleDateFormat) DateFormat.getTimeInstance(DateFormat.SHORT)).toPattern();
timeSpinner.setEditor(new JSpinner.DateEditor(timeSpinner, pattern));
```

Вы можете отображать произвольные последовательности в счетчике путем определения вашей собственной модели счетчика. В нашей программе мы имеем счетчик, который предлагает все возможные варианты перестановки букв в слове “meat”. Вы можете выбрать “mate”, “meta”, “team” и другие 20 вариантов перестановки, щелкая на кнопках счетчика.

При определении собственной модели необходимо расширять класс `AbstractSpinnerModel` и определять следующие четыре метода:

```
Object getValue()
void setValue(Object value)
Object getNextValue()
Object getPreviousValue()
```

Метод `getValue` должен вызывать метод `fireStateChanged` после установки нового значения. Метод `setValue` устанавливает новое значение. Он должен сгенерировать исключение `IllegalArgumentException`, если новое значение не является подходящим.



После установки нового значения метод `setValue` должен вызвать метод `fireStateChanged`. В противном случае поле счетчика не будет обновлено.

Методы `getNextValue` и `getPreviousValue` возвращают значения, которые должны следовать после или перед текущим значением, или `null`, если был достигнут конец последовательности.



Методы `getNextValue` и `getPreviousValue` не должны изменять текущее значение. Когда пользователь щелкает на кнопке счетчика со стрелкой вверх, происходит вызов метода `getNextValue`. Если возвращаемое значение будет отлично от `null`, оно будет задано посредством вызова метода `setValue`.

В примере программы мы используем стандартный алгоритм для определения следующего и предыдущего варианта перестановки букв. Детали алгоритма для нас не важны.

В листинге 6.14 показано, как генерируются различные типы счетчиков. Щелкните на кнопке OK, чтобы увидеть значения счетчика.

Листинг 6.14. Содержимое файла `SpinnerTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.text.*;
4.  import java.util.*;
5.  import javax.swing.*;
6.
7.  /**
8.     Программа для тестирования счетчиков.
9.  */
10. public class SpinnerTest
11. {
12.    public static void main(String[] args)
13.    {
14.       SpinnerFrame frame = new SpinnerFrame();
15.       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16.       frame.setVisible(true);
17.    }
18. }
19.
20. /**
21.    Фрейм с панелью, содержащей несколько счетчиков и кнопку,
22.    посредством которой отображаются значения счетчика.
23. */
```

```
24. class SpinnerFrame extends JFrame
25. {
26.    public SpinnerFrame()
27.    {
28.       setTitle("SpinnerTest");
29.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
30.       JPanel buttonPanel = new JPanel();
31.       okButton = new JButton("Ok");
32.       buttonPanel.add(okButton);
33.       dd(buttonPanel, BorderLayout.SOUTH);
34.
35.       mainPanel = new JPanel();
36.       mainPanel.setLayout(new GridLayout(0, 3));
37.       add(mainPanel, BorderLayout.CENTER);
38.
39.       JSpinner defaultSpinner = new JSpinner();
40.       addRow("Default", defaultSpinner);
41.
42.       JSpinner boundedSpinner = new JSpinner(new SpinnerNumberModel(5, 0, 10, 0.5));
43.       addRow("Bounded", boundedSpinner);
44.
45.       String[] fonts = GraphicsEnvironment
46.          .getLocalGraphicsEnvironment()
47.          .getAvailableFontFamilyNames();
48.
49.       JSpinner listSpinner = new JSpinner(new SpinnerListModel(fonts));
50.       addRow("List", listSpinner);
51.       JSpinner reverseListSpinner = new JSpinner(
52.          new
53.             SpinnerListModel(fonts)
54.             {
55.                public Object getNextValue()
56.                {
57.                   return super.getPreviousValue();
58.                }
59.                public Object getPreviousValue()
60.                {
61.                   return super.getNextValue();
62.                }
63.             });
64.       addRow("Reverse List", reverseListSpinner);
65.
66.       JSpinner dateSpinner = new JSpinner(new SpinnerDateModel());
67.       addRow("Date", dateSpinner);
68.
69.       JSpinner betterDateSpinner = new JSpinner(new SpinnerDateModel());
70.       String pattern = ((SimpleDateFormat) DateFormat.getDateInstance()).toPattern();
71.       betterDateSpinner.setEditor(new JSpinner.DateEditor(betterDateSpinner, pattern));
72.       addRow("Better Date", betterDateSpinner);
73.       JSpinner timeSpinner = new JSpinner(
74.          new SpinnerDateModel(
75.             new GregorianCalendar(2000, Calendar.JANUARY, 1, 12, 0, 0).getTime(),
76.                null, null, Calendar.HOUR));
77.       addRow("Time", timeSpinner);
78.       JSpinner permSpinner = new JSpinner(new PermutationSpinnerModel("meat"));
79.       addRow("Word permutations", permSpinner);
80.    }
```

```
81.    /**
82.     *  Добавляем строку в главную панель.
83.     *  @param labelText Метка счетчика
84.     *  @param spinner Пример счетчика
85.     */
86.    public void addRow(String labelText, final JSpinner spinner)
87.    {
88.       mainPanel.add(new JLabel(labelText));
89.       mainPanel.add(spinner);
90.       final JLabel valueLabel = new JLabel();
91.       mainPanel.add(valueLabel);
92.       okButton.addActionListener(new
93.          ActionListener()
94.          {
95.             public void actionPerformed(ActionEvent event)
96.             {
97.                Object value = spinner.getValue();
98.                valueLabel.setText(value.toString());
99.             }
100.         });
101.   }
102.
103.   public static final int DEFAULT_WIDTH = 400;
104.   public static final int DEFAULT_HEIGHT = 250;
105.
106.   private JPanel mainPanel;
107.   private JButton okButton;
108. }
109.
110. /**
111.    Модель, динамически генерирующая перестановки букв в слове.
112. */
113. class PermutationSpinnerModel extends AbstractSpinnerModel
114. {
115.    /**
116.     *  Создаем модель.
117.     *  @param w Слово, в котором будет производиться перестановка
118.     */
119.    public PermutationSpinnerModel(String w)
120.    {
121.       word = w;
122.    }
123.
124.    public Object getValue()
125.    {
126.       return word;
127.    }
128.
129.    public void setValue(Object value)
130.    {
131.       if (!(value instanceof String))
132.          throw new IllegalArgumentException();
133.       word = (String) value;
134.       fireStateChanged();
135.    }
136.
```

```
137.    public Object getNextValue()
138.    {
139.       int[] codePoints = toCodePointArray(word);
140.       for (int i = codePoints.length - 1; i > 0; i--)
141.       {
142.          if (codePoints[i - 1] < codePoints[i])
143.          {
144.             int j = codePoints.length - 1;
145.             while (codePoints[i - 1] > codePoints[j]) j--;
146.             swap(codePoints, i - 1, j);
147.             reverse(codePoints, i, codePoints.length - 1);
148.             return new String(codePoints, 0, codePoints.length);
149.          }
150.       }
151.       reverse(codePoints, 0, codePoints.length - 1);
152.       return new String(codePoints, 0, codePoints.length);
153.    }
154.    public Object getPreviousValue()
155.    {
156.       int[] codePoints = toCodePointArray(word);
157.       for (int i = codePoints.length - 1; i > 0; i--)
158.       {
159.          if (codePoints[i - 1] > codePoints[i])
160.          {
161.             int j = codePoints.length - 1;
162.             while (codePoints[i - 1] < codePoints[j]) j--;
163.             swap(codePoints, i - 1, j);
164.             reverse(codePoints, i, codePoints.length - 1);
165.             return new String(codePoints, 0, codePoints.length);
166.          }
167.       }
168.       reverse(codePoints, 0, codePoints.length - 1);
169.       return new String(codePoints, 0, codePoints.length);
170.    }
171.    private static int[] toCodePointArray(String str)
172.    {
173.       int[] codePoints = new int[str.codePointCount(0, str.length())];
174.       for (int i = 0, j = 0; i < str.length(); i++, j++)
175.       {
176.          int cp = str.codePointAt(i);
177.          if (Character.isSupplementaryCodePoint(cp)) i++;
178.          codePoints[j] = cp;
179.       }
180.       return codePoints;
181.    }
182.    private static void swap(int[] a, int i, int j)
183.    {
184.       int temp = a[i];
185.       a[i] = a[j];
186.       a[j] = temp;
187.    }
188.    private static void reverse(int[] a, int i, int j)
189.    {
190.       while (i < j) { swap(a, i, j); i++; j--; }
191.    }
192.    private String word;
193. }
```

API javax.swing.JSpinner 1.4

- `JSpinner()`

  Создает счетчик, редактирующий целое число, со стартовым числом 0, приращением 1, без ограничений.

- `JSpinner(SpinnerModel model)`

  Создает счетчик, использующий данную модель данных.

- `Object getValue()`

  Получает текущее значение счетчика.

- `void setValue(Object value)`

  Пытается установить значение счетчика. Генерирует исключение `IllegalArgumentException`, если модель не принимает это значение.

- `void setEditor(JComponent editor)`

  Устанавливает компонент, используемый для редактирования значения счетчика.

API 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, необходимо использовать константы `MIN_VALUE` и `MAX_VALUE` классов `Integer` и `Double`.

Параметры:	`initval`	Исходное значение
	`minimum`	Минимальное допустимое значение
	`maximum`	Максимальное допустимое значение
	`stepSize`	Увеличение или уменьшение каждого счетчика

API javax.swing.SpinnerListModel 1.4

- `SpinnerListModel(Object[] values)`
- `SpinnerListModel(List values)`

  Эти конструкторы позволяют создавать модели, выбирающие значение из данных значений.

API javax.swing.SpinnerDateModel 1.4

- `SpinnerDateModel()`

  Создает модель даты, в которой в качестве исходного значения принимается текущая дата, в которой нет верхних или нижних ограничений, а в качестве приращения принято `Calendar.DAY_OF_MONTH`.

- `SpinnerDateModel(Date initval, Comparable minimum, Comparable maximum, int step)`

Параметры:		
	`initval`	Исходное значение
	`minimum`	Минимальное допустимое значение, или `null`, если нужно убрать нижний предел
	`maximum`	Максимальное допустимое значение, или `null`, если нужно убрать верхний предел
	`step`	Поле даты для увеличения или уменьшения каждого счетчика; одна из констант 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`.

API java.text.SimpleDateFormat 1.1

- `String toPattern() 1.2`

  Получает шаблон редактирования для данного форматировщика даты. Обычно используется следующий шаблон: `"yyyy-MM-dd"`. Более подробно об этом шаблоне можно прочитать в документации к Java SE.

API javax.swing.JSpinner.DateEditor 1.4

- `DateEditor(JSpinner spinner, String pattern)`

  Создает редактор даты для счетчика.

Параметры:		
	`spinner`	Счетчик, которому принадлежит этот редактор
	`pattern`	Шаблон формата для связанного с ним `SimpleDateFormat`

API javax.swing.AbstractSpinnerModel 1.4

- `Object getValue()`

  Получает текущее значение модели.

- `void setValue(Object value)`

  Пытается установить новое значение для модели. Генерирует исключение `IllegalArgumentException`, если значение неприменимо. При переопределении этого метода необходимо вызывать `fireStateChanged` после установки нового значения.

- `Object getNextValue()`
- `Object getPreviousValue()`

  Вычисляет (но не устанавливает) следующее или предыдущее значение в последовательности, определенной моделью.

Отображение HTML-данных посредством панели JEditorPane

В отличие от текстовых компонентов, рассмотренных нами до настоящего момента, класс `JEditorPane` позволяет отображать и редактировать текст в формате HTML и RTF. (Формат Rich Text Format (RTF) используется в приложениях Microsoft для обмена документами. Этот формат не очень хорошо документирован и не всегда удовлетворительно работает даже в приложениях Microsoft. В настоящей книге формат RTF не рассматривается.)

Возможности редактора `JEditorPane` ограниченны. Объект отображения HTML-данных может воспроизводить простые файлы, но не справляется с более сложными языковыми конструкциями, которые часто можно встретить в Web. К тому же этот редактор работает не очень стабильно.

Оптимальным использованием компонента `JEditorPane`, вероятно, является отображение справочной информации, представленной в формате HTML. Поскольку текст справочной системы вы формируете самостоятельно, вы без труда обойдете все ограничения, характерные для компонента `JEditorPane`.



Для получения более подробной информации о профессиональных инструментах создания справочных систем следует обратиться по адресу: `http://java.sun.com/products/javahelp/index.html`.

В листинге 6.15 представлен код программы, окно которой содержит панель редактирования. Эта панель может отображать HTML-страницу. Введите URL, начинающийся с `http:` или `file:`, в поле редактирования, а затем щелкните на кнопке Load (Загрузить). После этого в панели редактирования будет отображена заданная страница (рис. 6.39).

В этом окне все гипертекстовые ссылки активны, т.е. после щелчка на любой ссылке приложение загрузит указанную Web-страницу. Причем щелчок на кнопке Back (Назад) приведет к выводу предыдущей страницы.

Фактически эта программа представляет собой простейший Web-браузер. Конечно, в нем не поддерживаются специальные возможности, например кэширование страниц или список использованных ссылок, которые присутствуют в коммерческих Web-браузерах. Кроме того, эта программа не способна отображать аплеты.

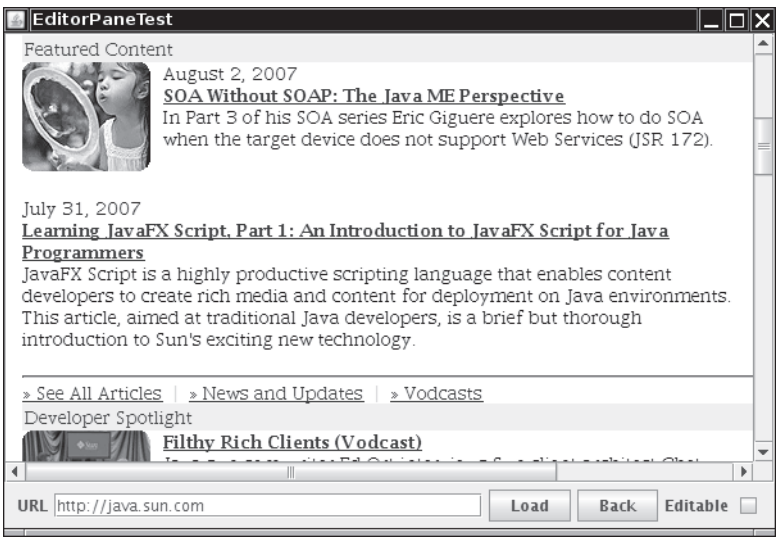


Рис. 6.39. Панель редактирования, отображающая HTML-страницу

После установки флажка Editable (Редактируемый) разрешается редактировать содержимое этой панели. Теперь в ней можно вводить текст и использовать клавишу <Backspace> для удаления символов. Этот компонент также может обрабатывать стандартные комбинации клавиш <Ctrl+X>, <Ctrl+C> и <Ctrl+V>, которые выполняют функ-

ции вырезания, копирования и вставки текста. Однако чтобы реализовать поддержку шрифтов и форматирование, придется затратить немалые усилия.

При редактировании компонента гипертекстовые ссылки не активны. Кроме того, в режиме редактирования в некоторых Web-страницах будут видны команды JavaScript, комментарии и другие дескрипторы (рис. 6.40). В этом примере режим редактирования разрешается использовать в учебных целях, но для реальных программ он не подходит.

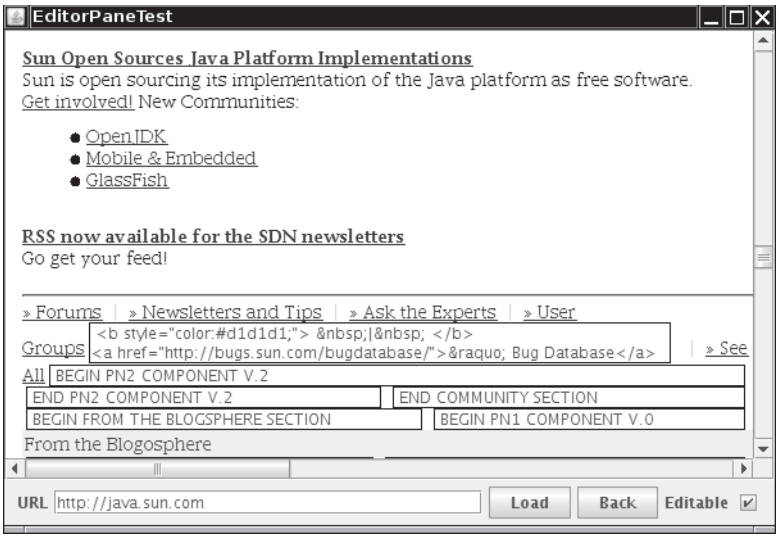


Рис. 6.40. Панель в режиме редактирования



По умолчанию компонент `JEditorPane` находится в режиме редактирования, а для отключения этого режима служит вызов `editorPane.setEditable(false)`.

Основные функции панели редактирования достаточно просты в использовании. Для загрузки документов предназначен метод `setPage()` с параметром в виде строки, содержащей URL, или объекта `URL`:

```
JEditorPane editorPane = new JEditorPane();
editorPane.setPage(url);
```

Класс `JEditorPane` расширяет класс `JTextComponent`, поэтому для отображения обычного текста можно вызывать метод `setText()` суперкласса.



Из документации на API трудно понять, загружает метод `setPage()` новый документ в отдельном потоке или нет. (Большинство программистов приветствовали бы использование для этой цели отдельного потока; класс `JEditorPane` нельзя назвать сверхбыстродействующим.) Для того чтобы явно указать на необходимость работы в специальном потоке, можно использовать следующий фрагмент кода:

```
AbstractDocument doc = (AbstractDocument) editorPane.getDocument();
doc.setAsynchronousLoadPriority(0);
```

Для обработки щелчков мыши на гипертекстовых ссылках следует создать обработчик HyperlinkListener, который содержит единственный метод hyperlinkUpdate() с параметром типа HyperlinkEvent. Этот метод вызывается при перемещениях мыши над ссылками и щелчках кнопкой мыши на ссылках.

Для определения типа события нужно вызвать метод getEventType(). Он возвращает одно из перечисленных ниже значений:

```
HyperlinkEvent.EventType.ACTIVATED
HyperlinkEvent.EventType.ENTERED
HyperlinkEvent.EventType.EXITED
```

Значение HyperlinkEvent.EventType.ACTIVATED соответствует щелчку кнопкой мыши на гипертекстовой ссылке. Пользователь поступает так, например, для открытия указанной Web-страницы. 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 помещает на нее курсор мыши.



Остается загадкой, почему разработчики Swing не создали три отдельных метода для управления этими событиями в интерфейсе HyperlinkListener.

Метод getURL() класса HyperlinkEvent возвращает URL, соответствующий данной гипертекстовой ссылке. Ниже представлен способ создания обработчика для активизированных пользователем ссылок:

```
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);
            }
         }
      }
   });
```

Обработчик события извлекает URL и обновляет панель редактирования. Метод setPage() может генерировать исключение IOException. В этом случае отображается сообщение об ошибке. Оно представляется обычным текстом.

В листинге 6.15 приводится пример использования основных средств панели редактирования, который необходим для создания справочной системы, отображающей текст в формате HTML. Компонент JEditorPane имеет более сложную структуру, чем таблица или дерево. Однако если на его основе не требуется создавать специальные текстовые редакторы и объекты отображения специальных форматов текста, то все сложные процессы будут скрыты от программиста.

Листинг 6.15. Содержимое файла `EditorPaneTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.io.*;
4.  import java.util.*;
5.  import javax.swing.*;
6.  import javax.swing.event.*;
7.
8.  /**
9.   * Пример отображения HTML-документов в панели редактирования.
10.  * @version 1.03 2007-08-01
11.  * @author Cay Horstmann
12.  */
13. public class EditorPaneTest
14. {
15.    public static void main(String[] args)
16.    {
17.       EventQueue.invokeLater(new Runnable()
18.          {
19.             public void run()
20.             {
21.                JFrame frame = new EditorPaneFrame();
22.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.                frame.setVisible(true);
24.             }
25.          });
26.    }
27. }
28.
29. /**
30.  * Фрейм, содержащий панель редактирования, поле
31.  * редактирования, кнопку для подтверждения ввода
32.  * URL и загрузки документа, а также кнопку Back для
33.  * возврата к документу, который просматривался ранее.
34.  */
35. class EditorPaneFrame extends JFrame
36. {
37.    public EditorPaneFrame()
38.    {
39.       setTitle("EditorPaneTest");
40.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
41.
42.       final Stack<String> urlStack = new Stack<String>();
43.       final JEditorPane editorPane = new JEditorPane();
44.       final JTextField url = new JTextField(30);
45.
46.       // Установка обработчика гипертекстовых ссылок.
47.
48.       editorPane.setEditable(false);
49.       editorPane.addHyperlinkListener(new HyperlinkListener()
50.          {
51.             public void hyperlinkUpdate(HyperlinkEvent event)
52.             {
53.                if (event.getEventType() ==
54.                   HyperlinkEvent.EventType.ACTIVATED)
55.                {
```

```
56.                try
57.                {
58.                   // Запоминание URL на случай активизации
59.                   // кнопки Back.
60.                   urlStack.push(event.getURL().toString());
61.                   // Отображение документа, соответствующего
62.                   // URL в поле редактирования.
63.                   url.setText(event.getURL().toString());
64.                   editorPane.setPage(event.getURL());
65.                }
66.                catch (IOException e)
67.                {
68.                   editorPane.setText("Exception: " + e);
69.                }
70.             }
71.          }
72.       });
73.
74.       // Установка флажков опций для управления
75.       // режимом редактирования.
76.
77.       final JCheckBox editable = new JCheckBox();
78.       editable.addActionListener(new ActionListener()
79.          {
80.             public void actionPerformed(ActionEvent event)
81.             {
82.                editorPane.setEditable(editable.isSelected());
83.             }
84.          });
85.
86.       // Настройка кнопки для загрузки документов.
87.
88.       ActionListener listener = new ActionListener()
89.          {
90.             public void actionPerformed(ActionEvent event)
91.             {
92.                try
93.                {
94.                   // Запоминание URL на случай активизации
95.                   // кнопки Back.
96.                   urlStack.push(url.getText());
97.                   editorPane.setPage(url.getText());
98.                }
99.                catch (IOException e)
100.               {
101.                  editorPane.setText("Exception: " + e);
102.               }
103.            }
104.         };
105.
106.      JButton loadButton = new JButton("Load");
107.      loadButton.addActionListener(listener);
108.      url.addActionListener(listener);
109.
110.      // Настройка кнопки Back.
111.
112.      JButton backButton = new JButton("Back");
```

```
113.      backButton.addActionListener(new ActionListener()
114.        {
115.          public void actionPerformed(ActionEvent event)
116.          {
117.            if (urlStack.size() <= 1) return;
118.            try
119.            {
120.               // Извлечение запомненного URL.
121.               urlStack.pop();
122.               // Отображение документа, соответствующего
123.               // URL в поле редактирования.
124.               String urlString = urlStack.peek();
125.               url.setText(urlString);
126.               editorPane.setPage(urlString);
127.            }
128.            catch (IOException e)
129.            {
130.               editorPane.setText("Exception: " + e);
131.            }
132.          }
133.        });
134.
135.      add(new JScrollPane(editorPane), BorderLayout.CENTER);
136.
137.      // Включение всех компонентов в состав панели.
138.
139.      JPanel panel = new JPanel();
140.      panel.add(new JLabel("URL"));
141.      panel.add(url);
142.      panel.add(loadButton);
143.      panel.add(backButton);
144.      panel.add(new JLabel("Editable"));
145.      panel.add(editable);
146.
147.      add(panel, BorderLayout.SOUTH);
148.    }
149.
150.    private static final int DEFAULT_WIDTH = 600;
151.    private static final int DEFAULT_HEIGHT = 400;
152. }
```

API javax.swing.JEditorPane 1.2

- `void setPage(URL url)`
  Загружает в панель редактирования Web-страницу, указанную посредством параметра `url`.
- `void addHyperlinkListener(HyperlinkListener listener)`
  Связывает с панелью редактирования обработчик гипертекстовых ссылок.

API javax.swing.event.HyperlinkListener 1.2

- `void hyperlinkUpdate(HyperlinkEvent event)`
  Вызывается при каждом выборе гипертекстовой ссылки.

API javax.swing.HyperlinkEvent 1.2

- `URL getURL()`

  Возвращает URL, соответствующий выбранной гипертекстовой ссылке.

Получение информации о выполнении задачи

В следующих разделах обсуждаются три класса, предназначенные для отображения хода выполнения задач, решение которых занимает длительное время. `JProgressBar` — это компонент Swing, который визуально представляет ход процесса. `ProgressMonitor` поддерживает диалоговое окно, в котором находится индикатор хода процесса, отображаемый посредством объекта `JProgressBar`. Элемент `ProgressMonitorInputStream` отображает диалоговое окно мониторинга процесса при чтении данных из потока.

Индикатор хода процесса

Индикатор хода процесса (progress bar) — это прямоугольник, частично заполненный другим цветом. Он дает представление о том, какая часть вычислений, необходимых для решения задачи, уже выполнена. По умолчанию в составе прямоугольника также отображается строка в формате *n*%, где *n* — число. Пример окна, в нижней части которого содержится индикатор хода процесса, приведен на рис. 6.41.



Рис. 6.41. Индикатор хода процесса

Индикатор хода процесса создается подобно линейному регулятору; конструктору передается минимальное и максимальное значение, а также необязательное указание ориентации:

```
progressBar = new JProgressBar(0, 1000);
progressBar = new JProgressBar(SwingConstants.VERTICAL, 0, 1000);
```

Минимальное и максимальное значения можно также задать посредством методов `setMinimum()` и `setMaximum()`.

В отличие от линейного регулятора, индикатор хода процесса не реагирует на действия пользователя. Для изменения его состояния необходимо вызвать из программы метод `setValue()`.

Если вы вызовете приведенный ниже метод, то индикатор вычислит процент выполнения задачи и отобразит строку *n*%. Если вам нужно вывести другую последовательность символов, необходимо передать соответствующую строку методу `setString()`:

```
if (progressBar.getValue() > 900)
  progressBar.setString("Almost Done");
```

В программе, код которой приведен в листинге 6.16, рассматриваемый здесь компонент применяется для отображения хода выполнения задачи. Программа имитирует задачу, требующую длительного времени.

Класс `SimulatedActivity` увеличивает значение переменной `current` 10 раз в секунду. Когда оно достигает заранее обусловленной величины (переменная `target`), выполнение этого действия завершается. Мы используем класс `SwingWorker`, чтобы реализовать это действие и обновить компонент хода выполнения задачи в методе `process`. `SwingWorker` вызывает метод из потока диспетчеризации событий, поэтому мы спокойно можем обновлять этот компонент. (В главе 14 первого тома вы можете узнать больше о безопасности потоков в Swing.)

В JDK 1.4 был реализован индикатор, уведомляющий о самом факте выполнения задачи. Он обеспечивает вывод анимационного изображения, но не отображает никаких сведений, позволяющих судить о том, какая часть задачи выполнена. Такого рода индикаторы часто применяются в браузерах. Они сообщают, что браузер ожидает ответа от сервера, но не дают представления о том, сколько это ожидание может продлиться. Для отображения такого индикатора "неопределенного ожидания" нужно вызвать метод `setIndeterminate()`.

Код программы, рассматриваемой в качестве примера, приведен в листинге 6.16.

Листинг 6.16. Содержимое файла `ProgressBarTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.util.List*;
4.
5.  import javax.swing
6.
7.  /**
8.   * Данная программа демонстрирует использование
9.   * индикатора хода процесса для отображение сведений
10.  * о ходе выполнения задачи.
11.  * @version 1.04 2007-08-01
12.  * @author Cay Horstmann
13.  */
14. public class ProgressBarTest
15. {
16.    public static void main(String[] args)
17.    {
18.       EventQueue.invokeLater(new Runnable()
19.          {
20.             public void run()
21.             {
22.                JFrame frame = new ProgressBarFrame();
23.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24.                frame.setVisible(true);
25.             }
26.          });
27.    }
28. }
29.
30. /**
31.  * Фрейм, содержащий кнопку для запуска имитатора задачи,
32.  * индикатор хода процесса и текстовую область для вывода
33.  * данных о выполнении задачи.
34.  */
```

```
35. class ProgressBarFrame extends JFrame
36. {
37.    public ProgressBarFrame()
38.    {
39.       setTitle("ProgressBarTest");
40.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
41.
42.       // В этой текстовой области отображаются
43.       // данные, предоставляемые имитатором задачи.
44.       textArea = new JTextArea();
45.
46.       // Создание панели и помещение в нее кнопки
47.       // и индикатора хода процесса.
48.
49.       final int MAX = 1000;
50.       JPanel panel = new JPanel();
51.       startButton = new JButton("Start");
52.       progressBar = new JProgressBar(0, MAX);
53.       progressBar.setStringPainted(true);
54.       panel.add(startButton);
55.       panel.add(progressBar);
56.
57.       checkBox = new JCheckBox("indeterminate");
58.       checkBox.addActionListener(new ActionListener()
59.          {
60.             public void actionPerformed(ActionEvent event)
61.             {
62.                progressBar.setIndeterminate (checkBox.isSelected());
63.                 progressBar.setStringPainted(!progressBar.isIndeterminate());
64.             }
65.          });
66.       panel.add(checkBox);
67.       add(new JScrollPane(textArea), BorderLayout.CENTER);
68.       add(panel, BorderLayout.SOUTH);
69.
70.       // Создание обработчика для кнопки.
71.
72.       startButton.addActionListener(new ActionListener()
73.          {
74.             public void actionPerformed(ActionEvent event)
75.             {
76.                startButton.setEnabled(false);
77.                activity = new SimulatedActivity(MAX);
78.                activity.execute();
79.             }
80.          });
81.
82.    }
83.    private JButton startButton;
84.    private JProgressBar progressBar;
85.    private JCheckBox checkBox;
86.    private JTextArea textArea;
87.    private SimulatedActivity activity;
88.
89.    public static final int DEFAULT_WIDTH = 400;
90.    public static final int DEFAULT_HEIGHT = 200;
91.
```

```
92.    class SimulatedActivity extends SwingWorker<Void, Integer>
93.    {
94.       /**
95.        * Создание объекта имитатора. Поток организует счетчик
96.        * от 0 до заранее заданного значения.
97.        * @param t Величина, до которой увеличивается значение
98.        * счетчика.
99.        */
100.      public SimulatedActivity(int t)
101.      {
102.        current = 0;
103.        target = t;
104.     }
105.
106.     protected Void doInBackground() throws Exception
107.     {
108.        try
109.        {
110.           while (current < target)
111.           {
112.              Thread.sleep(100);
113.              current++;
114.              publish(current);
115.           }
116.        }
117.        catch(InterruptedException e)
118.        {
119.        }
120.     }
121.
122.     protected void process(List<Integer> chunks)
123.     {
124.        for (Integer chunk : chunks)
125.        {
126.           textArea.append(chunk + "\n");
127.           progressBar.setValue(chunk);
128.        }
129.     }
130.
131.     protected void done()
132.     {
133.        startButton.setEnabled(true);
134.     }
135.
136.     private int current;
137.     private int target;
138.   }
139. }
```

Мониторы задач

Индикатор хода процесса — простой компонент, помещаемый непосредственно в окно. Класс `ProgressMonitor` реализует достаточно сложное диалоговое окно, содержащее индикатор хода процесса (рис. 6.42). В этом окне есть кнопка Cancel (Отмена). Если пользователь щелкнет на ней, диалоговое окно монитора будет закрыто.

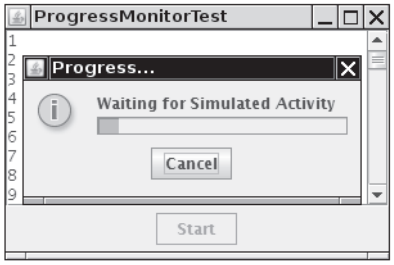


Рис. 6.42. Диалоговое окно монитора задачи

Кроме того, программа, выполнение которой отслеживается монитором, может проверять, закрыто ли диалоговое окно и, если пользователь активизировал кнопку Cancel, завершить свою работу. (Обратите внимание, что имя класса, реализующего данный компонент, начинается не с буквы `J`.)

При создании монитора задачи конструктору `ProgressMonitor` передаются следующие сведения:

- родительский компонент, поверх которого должно отображаться диалоговое окно;
- объект (строка пиктограмма или компонент), отображаемый в диалоговом окне;
- необязательное примечание, которое выводится под объектом;
- минимальное и максимальное значения.

Монитор не может самостоятельно определить, какая часть задачи выполнена, или завершить ее работу. Разработчик должен периодически обновлять состояние окна, вызывая метод `setProgress()`. (Этот метод выполняет те же функции, что и метод `setValue()` класса `JProgressBar`.) Если необходимость в мониторе задачи отпала, нужно вызвать метод `close()`, удалив диалоговое окно с экрана. Впоследствии вы можете использовать то же окно, вызвав метод `start()`.

Самой большой проблемой при использовании диалогового окна наблюдения за процессом является обработка запросов отмены. Невозможно присоединить обработчик события к кнопке Cancel. Вместо этого, необходимо периодически вызывать метод `isCanceled`, чтобы узнать, щелкнул ли пользователь программы на кнопке Cancel.

Если ваш рабочий поток может входить в состояние блокировки на неопределенный срок (например, при чтении введенных данных из сетевого соединения), он не может наблюдать за кнопкой Cancel. В программе мы продемонстрируем, как используется таймер для этой цели. Кроме того, мы возложим на таймер ответственность за обновление измерений хода выполнения.

Если вы запустите программу, код которой приведен в листинге 16.17, вы заметите интересную особенность. Диалоговое окно не исчезает немедленно по завершении задачи. Программе нужно убедиться, что имитатор задачи действительно окончил свою работу, либо завершить ее за время, меньшее времени, необходимого для отображения диалогового окна.

Управление временем происходит следующим образом. С помощью метода `setMillisToDecideToPopup()` задается число миллисекунд, которые должны пройти между созданием объекта диалогового окна и решением, следует ли отображать его. По умолчанию принимается значение 500 миллисекунд. Метод `setMillisToPopup()` предназначен для оценки времени, необходимого для отображения диалогового окна.

Разработчики Swing установили это значение равным по умолчанию 2 секундам. Они учли тот факт, что диалоговые окна Swing не появляются так быстро, как хотелось бы. Без необходимости изменять это значение не следует.

Листинг 6.17. Содержимое файла `ProgressMonitorTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.
4.  import javax.swing.*;
5.
6.  /**
7.   * Программа, демонстрирующая использование монитора задачи.
8.   * @version 1.04 2007-08-01
9.   * @author Cay Horstmann
10.  */
11. public class ProgressMonitorTest
12. {
13.    public static void main(String[] args)
14.    {
15.       EventQueue.invokeLater(new Runnable()
16.          {
17.             public void run()
18.             {
19.                JFrame frame = new ProgressMonitorFrame();
20.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.                frame.setVisible(true);
22.             }
23.          });
24.    }
25. }
26.
27. /**
28.  * Фрейм, содержащий кнопку, предназначенную для
29.  * запуска имитатора задачи, и текстовую область,
30.  * в которой отображаются сведения о работе имитатора.
31.  */
32. class ProgressMonitorFrame extends JFrame
33. {
34.    public ProgressMonitorFrame()
35.    {
36.       setTitle("ProgressMonitorTest");
37.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
38.
39.       // Текстовая область для вывода информации,
40.       // связанной с работой интерпретатора.
41.       textArea = new JTextArea();
42.
43.       // Панель с кнопкой.
44.       JPanel panel = new JPanel();
45.       startButton = new JButton("Start");
46.       panel.add(startButton);
47.
48.       add(new JScrollPane(textArea), BorderLayout.CENTER);
49.       add(panel, BorderLayout.SOUTH);
50.
```

```
51.       // Обработчик событий для кнопки.
52.       startButton.addActionListener(new ActionListener()
53.          {
54.             public void actionPerformed(ActionEvent event)
55.             {
56.                startButton.setEnabled(false);
57.                final int MAX = 1000;
58.
59.                // Запуск имитатора.
60.                activity = new SimulatedActivity(MAX);
61.                activity.execute();
62.
63.                // Запуск монитора задачи.
64.                progressDialog = new ProgressMonitor(ProgressMonitorFrame.this,
65.                      "Waiting for Simulated Activity", null, 0, MAX());
66.                cancelMonitor.start();
67.             }
68.          });
69.       // Обработчик событий для таймера.
70.       cancelMonitor = new Timer(500, new ActionListener()
71.          {
72.             public void actionPerformed(ActionEvent event)
73.             {
74.                if (progressDialog.isCanceled())
75.                {
76.                   activity.cancel(true);
77.                   startButton.setEnabled(true);
78.                }
79.                else if (activity.isDone())
80.                {
81.                   progressDialog.close();
82.                   startButton.setEnabled(true);
83.                }
84.                else
85.                {
86.                   progressDialog.setProgress(activity.getProgress());
87.                }
88.             }
89.          });
90.    }
91.    private Timer cancelMonitor;
92.    private JButton startButton;
93.    private ProgressMonitor progressDialog;
94.    private JTextArea textArea;
95.    private SimulatedActivity activity;
96.
97.    public static final int DEFAULT_WIDTH = 300;
98.    public static final int DEFAULT_HEIGHT = 200;
99.
100.   class SimulatedActivity extends SwingWorker<Void, Integer>
101.   {
102.      /**
103.       * Создание потока для имитатора выполнения задачи.
104.       * В потоке происходит увеличение значения счетчика
105.       * от 0 до заранее оговоренной величины.
106.       * @param t Конечное значение счетчика.
107.       */
```

```
108.      public SimulatedActivity(int t)
109.      {
110.         current = 0;
111.         target = t;
112.      }
113.      protected Void doInBackground() throws Exception
114.      {
115.         try
116.         {
117.            while (current < target)
118.            {
119.               Thread.sleep(100);
120.               current++;
121.               textArea.append(current + "\n");
122.               setProgress(current);
123.            }
124.         }
125.         catch(InterruptedException e)
126.         {
127.         }
128.         return null;
129.      }
130.      private int current;
131.      private int target;
132.   }
133. }
```

Отслеживание чтения данных из потока

В пакете Swing содержится фильтр `ProgressMonitorInputStream`, который автоматически отображает диалоговое окно с информацией о том, какая часть данных была уже прочитана из потока.

Использовать данный фильтр чрезвычайно просто. Следует лишь поместить `ProgressMonitorInputStream` в состав последовательности используемых потоков (подробно потоки рассматриваются в главе 12 первого тома).

Предположим, например, что вам требуется читать текст из файла. Формирование последовательности начинается с потока `FileInputStream`:

```
FileInputStream in = new FileInputStream(f);
```

Обычно этот поток преобразуют в `InputStreamReader`:

```
InputStreamReader reader = new InputStreamReader(in);
```

Если же вы хотите отслеживать чтение данных из потока, нужно сначала преобразовать `FileInputStream` в `ProgressMonitorInputStream`:

```
ProgressMonitorInputStream progressIn =
   new ProgressMonitorInputStream(parent, caption, in);
```

Конструктору `ProgressMonitorInputStream` передается родительский компонент, заголовок и, конечно же, поток, предназначенный для отслеживания. Метод `read()` данного класса анализирует количество прочитанных байтов и обновляет содержимое диалогового окна.

Создав объект `ProgressMonitorInputStream`, можно продолжить формирование последовательности потоков:

```
InputStreamReader reader = new InputStreamReader(progressIn);
```

На этом действия по организации мониторинга чтения данных заканчиваются. При чтении из файла автоматически отобразится окно, показанное на рис. 6.43.

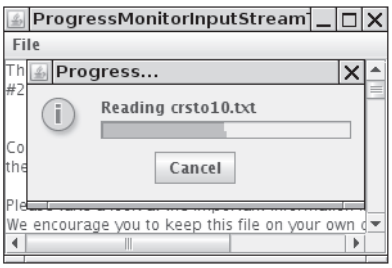


Рис. 6.43. Окно, позволяющее отслеживать процесс чтения данных из потока ввода



Для определения общего количества байтов в потоке объект `ProgressMonitorInputStream` использует метод `available()` класса `InputStream`. Однако этот метод возвращает лишь число байтов, которые могут быть прочитаны *без блокировки ввода*. Таким образом, рассматриваемый здесь фильтр хорошо работает с файлами и данными, получаемыми посредством протокола HTTP, поскольку их размер известен заранее, однако следует помнить, что результаты его применения к другим потокам могут быть гораздо хуже.

Программа, код которой приведен в листинге 6.18, подсчитывает строки в файле. Если размеры файла велики, отображается диалоговое окно, отображающее ход выполнения задачи.

Если пользователь щелкает на кнопке Cancel, поток ввода закрывается. Поскольку код, обрабатывающий ввод, уже знает о том, что нужно делать при окончании ввода, для обработки отмены не нужна дополнительная логика.

Обратите внимание, что в данном примере использован не самый эффективный способ заполнения текстовой области. Программа работала бы гораздо быстрее, если бы мы сначала прочитали файл в `StringBuilder`, а затем связали текстовую область с контекстом. Однако в данном случае медленное выполнение программы является преимуществом, так как главная задача примера — продемонстрировать индикацию хода выполнения задачи.

Чтобы избежать мерцания, мы не отображаем текстовую область до ее заполнения.

Листинг 6.18. Содержимое файла `ProgressMonitorInputStreamTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.io.*;
4.  import java.util.*;
5.  import javax.swing.*;
6.
7.  /**
8.   * Программа, демонстрирующая мониторинг процесса
9.   * чтения данных из файла.
10.  * @version 1.04 2007-08-01
11.  * @author Cay Horstmann
12.  */
```

```
13. public class ProgressMonitorInputStreamTest
14. {
15.    public static void main(String[] args)
16.    {
17.       EventQueue.invokeLater(new Runnable()
18.          {
19.             public void run()
20.             {
21.                JFrame frame = new LongListFrame();
22.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.                frame.setVisible(true);
24.             }
25.          });
26.    }
27. }
28. /**
29.  * Фрейм, содержащий меню, которое позволяет задавать
30.  * команду загрузки текстового файла, и текстовую область,
31.  * предназначенную для отображения содержимого файла.
32.  * Текстовая область создается после загрузки файла и
33.  * по окончании загрузки файла устанавливается в качестве
34.  * области содержимого фрейма. Это позволяет избежать
35.  * мерцания в процессе загрузки файла.
36.  */
37. class TextFrame extends JFrame
38. {
39.    public TextFrame()
40.    {
41.       setTitle("ProgressMonitorInputStreamTest");
42.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
43.
44.       textArea = new JTextArea();
45.       add(new JScrollPane(textArea));
46.
47.       chooser = new JFileChooser();
48.       chooser.setCurrentDirectory(new File(“.”));
49.
50.       JMenuBar menuBar = new JMenuBar();
51.       setJMenuBar(menuBar);
52.       JMenu fileMenu = new JMenu("File");
53.       menuBar.add(fileMenu);
54.       openItem = new JMenuItem("Open");
55.       openItem.addActionListener(new ActionListener()
56.          {
57.             public void actionPerformed(ActionEvent event)
58.             {
59.                try
60.                {
61.                   openFile();
62.                }
63.                catch(IOException exception)
64.                {
65.                   exception.printStackTrace();
66.                }
67.             }
68.          });
69.
```

```
70.       fileMenu.add(openItem);
71.       exitItem = new JMenuItem("Exit");
72.       exitItem.addActionListener(new ActionListener()
73.          {
74.             public void actionPerformed(ActionEvent event)
75.             {
76.                System.exit(0);
77.             }
78.          });
79.       fileMenu.add(exitItem);
80.    }
81.
82.    /**
83.     * Выбор файла для ввода, загрузка файла в текстовую
84.     * область и установка текстовой области в качестве
85.     * области содержимого фрейма.
86.     */
87.    public void openFile() throws IOException
88.    {
89.       int r = chooser.showOpenDialog(this);
90.       if (r != JFileChooser.APPROVE_OPTION) return;
91.       final File f = chooser.getSelectedFile();
92.
93.       // Создание последовательности потоков
94.       // и включение в нее фильтра.
95.
96.       FileInputStream fileIn = new FileInputStream(f);
97.       ProgressMonitorInputStream progressIn =
98.         new ProgressMonitorInputStream(this, "Reading " + f.getName(), fileIn);
99.       final Scanner in = new Scanner(progressIn);
100.
101.      textArea.setText("");
102.
103.      SwingWorker<Void, Void> worker = new SwingWorker<Void, Void>()
104.         {
105.            protected Void doInBackground() throws Exception
106.            {
107.               while (in.hasNextLine())
108.               {
109.                  String line = in.nextLine();
110.                  textArea.append(line);
111.                  textArea.append("\n");
112.               }
113.               in.close();
114.               return null;
115.            }
116.         };
117.      worker.execute();
118.   }
119.
120.   private JMenuItem openItem;
121.   private JMenuItem exitItem;
122.   private JTextArea textArea;
123.   private JFileChooser chooser;
124.
125.   public static final int DEFAULT_WIDTH = 300;
126.   public static final int DEFAULT_HEIGHT = 200;}
```

API javax.swing.JProgressBar 1.2

- `JProgressBar()`
- `JProgressBar(int direction)`
- `JProgressBar(int min, int max)`
- `JProgressBar(int direction, int min, int max)`

Создает индикатор хода процесса с указанной ориентацией, а также минимальным и максимальным значениями.

Параметры:		
	`direction`	Одна из констант `SwingConstants.HORIZONTAL` или `SwingConstants.VERTICAL`. По умолчанию принимается горизонтальная ориентация
	`min, max`	Минимальное и максимальное значения индикатора хода процесса. По умолчанию принимаются величины 0 и 100

- `int getMinimum()`
- `int getMaximum()`
- `void setMinimum(int value)`
- `void setMaximum(int value)`

Возвращает или устанавливает минимальное или максимальное значение.

- `int getValue()`
- `void setValue(int value)`

Возвращает или устанавливает текущее значение.

- `String getString()`
- `void setString(String s)`

Возвращает или устанавливает строку, отображаемую в полосе индикатора. Если вместо строки задано значение `null`, отображается строка по умолчанию *`n`*`%`.

- `boolean isStringPainted()`
- `void setStringPainted(boolean b)`

Возвращает или устанавливает значение свойства, управляющее отображением строки. Если значение данного свойства равно `true`, строка отображается в полосе индикатора. По умолчанию принимается значение `false`, запрещающее вывод строки.

- `boolean isIndeterminate()` 1.4
- `void setIndeterminate(boolean b)` 1.4

Возвращает или устанавливает значение свойства, управляющее "неопределенной" индикацией. Если значение данного свойства равно `true`, прямоугольник движется вперед и назад по полосе, указывая на то, что время ожидания неизвестно. По умолчанию принимается значение `false`.

API javax.swing.ProgressMonitor 1.2

- `ProgressMonitor(Component parent, Object message, String note, int min, int max)`

Создает диалоговое окно монитора задачи.

Параметры:	`parent`	Родительский компонент, поверх которого отображается диалоговое окно
	`message`	Объект сообщения, отображаемый в диалоговом окне
	`note`	Необязательная строка, выводимая под сообщением. Если значение данного параметра равно `null`, пространство для строки примечания не выделяется и последующие вызовы метода `setNote()` не дают никакого эффекта
	`min, max`	Минимальное и максимальное значения индикатора хода процесса

- `void setNote(String note)`
  Изменяет текст примечания.
- `void setProgress(int value)`
  Устанавливает заданное значение индикатора хода процесса.
- `void close()`
  Закрывает диалоговое окно.
- `boolean isCanceled()`
  Возвращает `true`, если пользователь закрыл диалоговое окно, щелкнув на кнопке Cancel.

API `javax.swing.ProgressMonitorInputStream 1.2`

- `ProgressMonitorInputStream(Component parent, Object message, InputStream in)`
  Создает фильтр входного потока и связывает с ним диалоговое окно, отслеживающее процесс чтения данных из потока.

Параметры:	`parent`	Родительский компонент, поверх которого должно отображаться диалоговое окно
	`message`	Объект сообщения, отображаемый в диалоговом окне
	`in`	Входной поток, чтение данных из которого отслеживается

Организаторы компонентов

Обсуждение расширенных средств библиотеки Swing закончим рассмотрением способов организации компонентов приложения. Для этого используются *панели с границей раздела* (split pane), которые разбивают область окна на несколько областей меньшего размера, *панели с вкладками* (tabbed pane), в которых пользователь, щелкая мышью, может переходить от одной вкладки к другой, а также *панель рабочего стола* (desktop pane), в которой можно размещать несколько *внутренних фреймов* (internal frame).

Панели с границей раздела

Использование панелей с границей раздела позволяет разбивать компонент на части, причем границы могут перемещаться. На рис. 6.44 показан пример разбивки области окна на три панели. Во внешней панели формируется горизонтальная граница, которая отделяет нижнюю панель с полем редактирования от верхней.

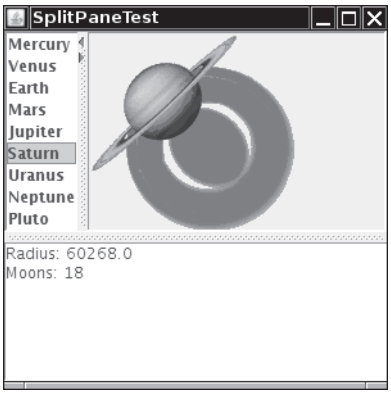


Рис. 6.44. Вложенные панели с границей раздела

Верхняя панель разделяется по вертикали; список планет слева отделяется от изображения выбранной планеты справа.

При создании панели с границей раздела нужно указать ориентацию границы (`JSplitPane.HORIZONTAL_SPLIT` или `JSplitPane.VERTICAL_SPLIT`) и два компонента, получаемых при разделении:

```
JSplitPane innerPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
  planetList, planetImage);
```

По желанию можно добавить пиктограммы полного развертывания панелей (one touch expand), которые показаны на рис. 6.44. Для стиля Metal они имеют вид треугольников. После щелчка на такой пиктограмме граница раздела переместится в указанном направлении до полного раскрытия панели.

Для использования этой функциональной возможности нужно вызвать метод

```
innerPane.setOneTouchExpandable(true);
```

Кроме того, можно использовать функцию постоянной перерисовки (continuous layout) содержимого панели по мере перемещения границы раздела:

```
innerPane.setContinuousLayout(true);
```

Результаты выглядят очень эффектно, но выполнение существенно замедляется. В данном примере эта возможность для горизонтальной границы раздела не используется, а потому содержимое нижней панели будет перерисовано только после перемещения границы.

Программа, код которой приведен в листинге 6.19, создает список с именами планет. Если пользователь выбирает какую-то планету, то в панели справа отображается ее рисунок, а в панели снизу — ее описание. После запуска программы попробуйте перемещать границы раздела и проверьте, как работают функции полного развертывания и постоянной перерисовки.

Листинг 6.19. Содержимое файла `SplitPaneTest.java`

```
1.  import java.awt.*;
2.  import javax.swing.*;
3.  import javax.swing.event.*;
4.
5.  /**
6.   * Использование панели с границей раздела
7.   * в качестве организатора компонентов.
```

```
8.    * @version 1.03 2007-08-01
9.    * @author Cay Horstmann
10.   */
11.  public class SplitPaneTest
12.  {
13.    public static void main(String[] args)
14.       {
15.          EventQueue.invokeLater(new Runnable()
16.             {
17.                public void run()
18.                {
19.                JFrame frame = new SplitPaneFrame();
20.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.                frame.setVisible(true);
22.             }
23.          });
24.    }
25.  }
26.  /**
27.   * Этот фрейм содержит две вложенных панели с границей
28.   * раздела. В них отображаются рисунки и текстовые данные.
29.   */
30.  class SplitPaneFrame extends JFrame
31.  {
32.    public SplitPaneFrame()
33.    {
34.       setTitle("SplitPaneTest");
35.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
36.
37.       // Создание компонентов для имен, изображений и описаний планет.
38.       final JList planetList = new JList(planets);
39.       final JLabel planetImage = new JLabel();
40.       final JTextArea planetDescription = new JTextArea();
41.
42.       planetList.addListSelectionListener(new ListSelectionListener()
43.          {
44.             public void valueChanged(ListSelectionEvent event)
45.             {
46.                Planet value = Planet) planetList.getSelectedValue();
47.
48.                // Обновление рисунка и описания.
49.
50.                planetImage.setIcon(value.getImage());
51.                planetDescription.setText(value.getDescription());
52.             }
53.          });
54.       // Установка панелей с границами раздела.
55.       JSplitPane innerPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
56.          planetList, planetImage);
57.
58.       innerPane.setContinuousLayout(true);
59.       innerPane.setOneTouchExpandable(true);
60.
61.       JSplitPane outerPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
62.          innerPane, planetDescription);
63.       add(outerPane, BorderLayout.CENTER);
64.    }
```

```
65.    private Planet[] planets = { new Planet("Mercury", 2440, 0),
66.          new Planet("Venus", 6052, 0), new Planet("Earth", 6378, 1),
67.          new Planet("Mars", 3397, 2), new Planet("Jupiter", 71492, 16),
68.          new Planet("Saturn", 60268, 18), new Planet("Uranus", 25559, 17),
69.          new Planet("Neptune", 24766, 8), new Planet("Pluto", 1137, 1), };
70.    private static final int DEFAULT_WIDTH = 300;
71.    private static final int DEFAULT_HEIGHT = 300;
72. }
73.
74. /**
75.  *  Описание планеты.
76.  */
77. class Planet
78. {
79.    /**
80.     * Создание объекта, представляющего планету.
81.     * @param n Название планеты
82.     * @param r Радиус планеты
83.     * @param m Количество спутников
84.     */
85.    public Planet(String n, double r, int m)
86.    {
87.       name = n;
88.       radius = r;
89.       moons = m;
90.       image = new ImageIcon(name + ".gif");
91.    }
92.
93.    public String toString()
94.    {
95.       return name;
96.    }
97.
98.    /**
99.     * Получение описания планеты.
100.    * @return Описание
101.    */
102.   public String getDescription()
103.   {
104.      return "Radius: " + radius + "\nMoons: " + moons + "\n";
105.   }
106.
107.   /**
108.    * Получение изображения планеты.
109.    * @return Изображение
110.    */
111.   public ImageIcon getImage()
112.   {
113.      return image;
114.   }
115.
116.   private String name;
117.   private double radius;
118.   private int moons;
119.   private ImageIcon image;
120. }
```

API 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)`

Создает новую панель.

Параметры:		
	`direction`	Одно из значений: `HORIZONTAL_SPLIT` или `VERTICAL_SPLIT`
	`continuousLayout`	Значение `true`, если содержимое компонентов постоянно обновляется при перемещении границы раздела
	`first, second`	Добавляемые компоненты

- `boolean isOneTouchExpandable()`
- `void setOneTouchExpandable(boolean b)`

Возвращает или устанавливает значение свойства, управляющего полным развертыванием панели. Значения `true` разрешает полное развертывание панели по щелчку на соответствующей пиктограмме.

- `boolean isContinuousLayout()`
- `void setContinuousLayout(boolean b)`

Возвращает или устанавливает значение свойства постоянной перерисовки панели при перемещении границы раздела. Значение `true` этого свойства разрешает постоянную перерисовку панели.

- `void setLeftComponent(Component c)`
- `void setTopComponent(Component c)`

Производит тот же эффект, что и указание `c` в качестве первого компонента в панели с границей раздела.

- `void setRightComponent(Component c)`
- `void setBottomComponent(Component c)`

Производит тот же эффект, что и указание `c` в качестве второго компонента в панели с границей раздела.

Панели с вкладками

Панели с вкладками можно использовать для создания очень удобного пользовательского интерфейса. Сложное диалоговое окно разбивается на несколько частей, каждая из которых содержит группу связанных по смыслу параметров. Вкладки можно также применять для просмотра документов или рисунков, как показано на рис. 6.45.

Для создания панели с вкладками, прежде всего, потребуется создать объект `JTabbedPane`, а затем включить в него вкладки:

```
JTabbedPane tabbedPane = new JTabbedPane();
tabbedPane.addTab(title, icon, component);
```

Последний параметр метода `addTab()` имеет тип `Component`. Для включения нескольких компонентов в одну и ту же вкладку нужно сначала упаковать их в контейнер (например, `JPanel`).

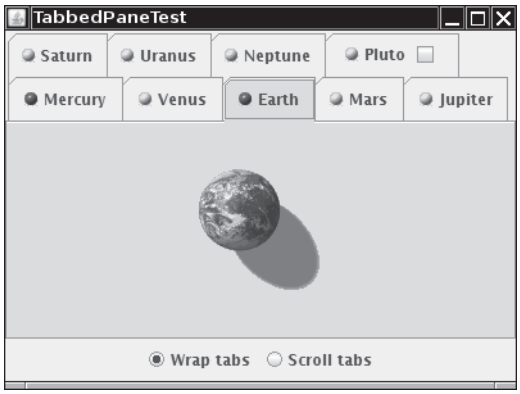


Рис. 6.45. Панель с вкладками

Пиктограмму `icon` можно не указывать; для этого предусмотрен другой вариант метода `addTab()`:

```
tabbedPane.addTab(title, component);
```

Очередную вкладку можно включить в нужном месте существующего набора вкладок. Для этого необходимо указать ее индекс:

```
tabbedPane.insertTab(title, icon, component, tooltip, index);
```

Для удаления вкладки из набора вкладок следует использовать метод `removeTabAt()`:

```
tabbedPane.removeTabAt(index);
```

Новая вкладка не будет отображена автоматически. Для этого придется выбрать ее с помощью метода `setSelectedIndex()`. Например, для отображения только что добавленной вкладки следует использовать приведенный ниже метод:

```
tabbedPane.setSelectedIndex(tabbedPane.getTabAcount() - 1);
```

Для отображения большого количества ярлыков вкладок может потребоваться большое пространство. Чтобы избежать этого, в JDK 1.4 предусмотрена возможность прокрутки ярлыков вкладок. При этом отображается только один ряд ярлыков и набор стрелок для прокрутки остальных (рис. 6.46).

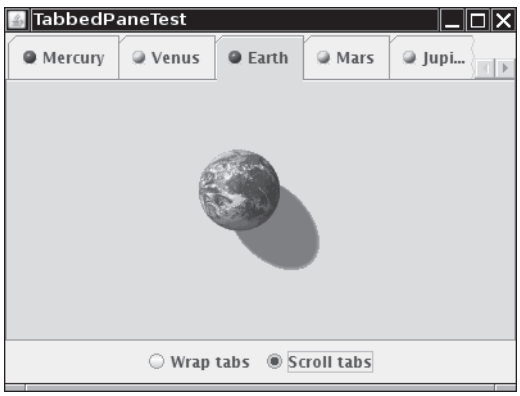


Рис. 6.46. Панель, поддерживающая прокрутку ярлыков вкладок

Для указания режима отображения ярлыков (вывод всех или использование прокрутки) применяются перечисленные ниже методы:

```
tabbedPane.setTabLayoutPolicy(JTabbedPane.WRAP_TAB_LAYOUT);
tabbedPane.setTabLayoutPolicy(JTabbedPane.SCROLL_TAB_LAYOUT);
```

Панели с вкладками могут иметь мнемонику, наподобие элементов меню. Например:

```
int marsIndex = tabbedPane.indexOfTab("Mars");
tabbedPane.setMnemonicAt(marsIndex, KeyEvent.VK_M);
```

Затем M подчеркивается, и пользователи программы могут выбрать вкладку посредством нажатия комбинации <Alt+M>.

В версии Java SE 6 вы можете добавлять произвольные компоненты в заголовки вкладок. Сначала нужно добавить вкладку, а затем вызвать следующий метод:

```
tabbedPane.setTabComponentAt(index, component);
```

В нашей программе мы добавляем окно закрытия на вкладку Pluto (поскольку, между прочим, некоторые астрономы не считают Плутон настоящей планетой). Это делается посредством установки компонента вкладки в панели, содержащей два компонента: метку со значком и текст вкладки, а также флажок со слушателем действия, удаляющий вкладку.

Далее рассматривается пример, демонстрирующий интересный способ использования панели с вкладками. Иногда перед отображением содержимого компонента требуется обновить его. В этом примере изображение планеты загружается только после щелчка кнопкой мыши на вкладке.

Для уведомления о таком событии нужно создать обработчик `ChangeListener` и связать его с панелью. Учтите, что этот обработчик создается для всей панели, а не для какого-либо компонента:

```
tabbedPane.addChangeListener(listener);
```

После щелчка кнопкой мыши на данной вкладке вызывается метод `stateChanged()` класса `ChangeListener`. При этом в качестве источника события выступает панель с вкладками. Для определения вкладки — источника события нужно использовать метод `getSelectedIndex()`:

```
public void stateChanged(ChangeEvent event)
{
   int n = tabbedPane.getSelectedIndex();
   loadTab(n);
}
```

В листинге 6.20 для всех компонентов вкладки сначала задается значение `null`. При выборе новой вкладки выполняется проверка на равенство `null` и подстановка изображения. (Это происходит мгновенно и незаметно для пользователя, который даже не успеет разглядеть пустую вкладку.) Кроме того, для демонстрации использования пиктограмм в заголовках, вкладки, которые просматривались, обозначаются красным кругом, а еще не просмотренные — желтым.

Листинг 6.20. Содержимое файла `TabbedPaneTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.
4.  import javax.swing.*;
5.  import javax.swing.event.*;
```

```
6.  /**
7.   * Пример организатора компонентов
8.   * на основе панели с вкладками.
9.   * @version 1.03 2007-08-01
10.  * @author Cay Horstmann
11.  */
12. public class TabbedPaneTest
13. {
14.    public static void main(String[] args)
15.    {
16.       EventQueue.invokeLater(new Runnable()
17.          {
18.             public void run()
19.             {
20.
21.                JFrame frame = new TabbedPaneFrame();
22.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.                frame.setVisible(true);
24.             }
25.          });
26.    }
27. }
28.
29. /**
30.  * Этот фрейм содержит панель с вкладками, а также переключатели
31.  * режима полного отображения и прокрутки ярлыков вкладок.
32.  */
33. class TabbedPaneFrame extends JFrame
34. {
35.    public TabbedPaneFrame()
36.    {
37.       setTitle("TabbedPaneTest");
38.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
39.
40.       tabbedPane = new JTabbedPane();
41.       // Указание значения null для компонентов, и
42.       // задержка их загрузки до первого обращения
43.       // к соответствующей вкладке.
44.
45.       ImageIcon icon = new ImageIcon("yellow-ball.gif");
46.
47.       tabbedPane.addTab("Mercury", icon, null);
48.       tabbedPane.addTab("Venus", icon, null);
49.       tabbedPane.addTab("Earth", icon, null);
50.       tabbedPane.addTab("Mars", icon, null);
51.       tabbedPane.addTab("Jupiter", icon, null);
52.       tabbedPane.addTab("Saturn", icon, null);
53.       tabbedPane.addTab("Uranus", icon, null);
54.       tabbedPane.addTab("Neptune", icon, null);
55.       tabbedPane.addTab("Pluto", icon, null);
56.
57.       final int plutoIndex = tabbedPane.indexOfTab("Pluto");
58.       JPanel plutoPanel = new JPanel();
59.       plutoPanel.add(new JLabel("Pluto", icon, SwingConstants.LEADING));
60.       JToggleButton plutoCheckBox = new JCheckBox();
```

```
61.       plutoCheckBox.addActionListener(new ActionListener()
62.       {
63.          public void actionPerformed(ActionEvent e)
64.          {
65.             tabbedPane.remove(plutoIndex);
66.          }
67.       });
68.       plutoPanel.add(plutoCheckBox);
69.       tabbedPane.setTabComponentAt(plutoIndex, plutoPanel);
70.
71.       add(tabbedPane, "Center");
72.
73.       tabbedPane.addChangeListener(new ChangeListener()
74.          {
75.             public void stateChanged(ChangeEvent event)
76.             {
77.
78.                // Проверка, установлено ли для компонента
79.                // значение null.
80.                if (tabbedPane.getSelectedComponent() == null)
81.                {
82.                   // Указание пиктограммы для заголовка.
83.
84.                   int n = tabbedPane.getSelectedIndex();
85.                   loadTab(n);
86.                }
87.             }
88.          });
89.       loadTab(0);
90.       JPanel buttonPanel = new JPanel();
91.       ButtonGroup buttonGroup = new ButtonGroup();
92.       JRadioButton wrapButton = new JRadioButton("Wrap tabs");
93.       wrapButton.addActionListener(new
94.          ActionListener()
95.          {
96.             public void actionPerformed(ActionEvent event)
97.             {
98.                tabbedPane.setTabLayoutPolicy
99.                   (JTabbedPane.WRAP_TAB_LAYOUT);
100.            }
101.         });
102.      buttonPanel.add(wrapButton);
103.      buttonGroup.add(wrapButton);
104.      wrapButton.setSelected(true);
105.      JRadioButton scrollButton = new JRadioButton("Scroll tabs");
106.      scrollButton.addActionListener(new ActionListener()
107.         {
108.            public void actionPerformed(ActionEvent event)
109.            {
110.               tabbedPane.setTabLayoutPolicy
111.                  (JTabbedPane.SCROLL_TAB_LAYOUT);
112.            }
113.         });
114.      buttonPanel.add(scrollButton);
115.      buttonGroup.add(scrollButton);
116.      add(buttonPanel, BorderLayout.SOUTH);
117.   }
```

```
118.    /**
119.     * Загрузка компонента и отображение на вкладке.
120.     * @param n Индекс вкладки
121.     */
122.    private void loadTab(int n)
123.    {
124.       String title = tabbedPane.getTitleAt(n);
125.       ImageIcon planetIcon = new ImageIcon(title + ".gif");
126.       tabbedPane.setComponentAt(n, new JLabel(planetIcon));
127.       // Вкладка помечается как уже просмотренная.
128.       tabbedPane.setIconAt(n, new ImageIcon("red-ball.gif"));
129.    }
130.    private JTabbedPane tabbedPane;
131.    private static final int DEFAULT_WIDTH = 400;
132.    private static final int DEFAULT_HEIGHT = 300;
133. }
```

API javax.swing.JTabbedPane 1.2

- `JTabbedPane()`
- `JTabbedPane(int placement)`

  Создает панель с вкладками.

  Параметр: `placement` Одно из значений: `SwingConstants.TOP`, `SwingConstants.LEFT`, `SwingConstants.RIGHT`, `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)`

  Добавляет вкладку в конец набора, присутствующего в составе панели.

- `void insertTab(String title, Icon icon, Component c, String tooltip, int index)`

  Помещает вкладку в позицию с указанным индексом.

- `void removeTab(int index)`

  Удаляет вкладку с указанным индексом из панели.

- `void setSelectedIndex(int index)`

  Выбирает вкладку с указанным индексом.

- `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)`

  Возвращает или устанавливает заголовок, пиктограмму или компонент вкладки с указанным индексом.

- `int indexOfTab(String title)`
- `int indexOfTab(Icon icon)`
- `int indexOfComponent(Component c)`

  Возвращает индекс вкладки с указанным заголовком, пиктограммой или компонентом.

- `int getTabCount()`

  Возвращает общее количество вкладок в составе панели.

- `int getTabLayoutPolicy()`
- `void setTabLayoutPolicy(int policy)` 1.4

  Возвращает или устанавливает способ отображения ярлыков вкладок. Они могут отображаться либо полностью, либо частично с возможностью прокрутки.

  Параметр: `policy` Одно из значений: `JtabbedPane.WRAP_TAB_LAYOUT` или `JTabbedPane.SCROLL_TAB_LAYOUT`

- `int getMnemonicAt(int index)` 1.4
- `void setMnemonicAt(int index, int mnemonic)`

  Возвращает или устанавливает мнемонический символ в данном индексе вкладки. Символ определяется как константа `VK_X` из класса `KeyEvent`. Значение `-1` обозначает, что это не мнемоника.

- `Component getTabComponentAt(int index)` 6
- `void setTabComponentAt(int index, Component c)` 6

  Возвращает или получает компонент, который отображает заголовок вкладки с данным индексом. Если этот компонент равен `null`, будут визуализироваться значок вкладки и заголовок. В противном случае на вкладке будет визуализироваться только данный компонент.

- `int indexOfTabComponent(Component c)` 6.0

  Возвращает индекс вкладки с данным компонентом заголовка.

- `void addChangeListener(ChangeListener listener)`

  Связывает с панелью обработчик, который уведомляется о выборе пользователем очередной вкладки.

Панель рабочего стола и внутренние фреймы

Во многих приложениях предлагаемая информация отображается в нескольких окнах, которые содержатся внутри большого фрейма. При свертывании фрейма все эти окна также скрываются. В операционной системе Windows такой пользовательский интерфейс называется *многодокументным интерфейсом* (multiple document interface — MDI). На рис. 6.47 показано типичное приложение, в котором используется такой интерфейс.

Ранее этот стиль представления пользовательского интерфейса был очень распространен, но в последнее время он стал менее популярным. В частности, многие браузеры для отображения Web-страниц используют несколько фреймов. Какой способ лучше? Оба подхода имеют преимущества и недостатки. MDI-интерфейс позволяет справиться с беспорядком, который возникает при наличии большого количества окон. Но при работе с несколькими окнами высокого уровня для организации переходов между ними придется использовать кнопки и горячие клавиши операционной системы.

В межплатформенном мире Java не предусмотрено использование каких-либо инструментов операционной системы. Поэтому пользовательский интерфейс на основе фреймов имеет гораздо больший смысл.

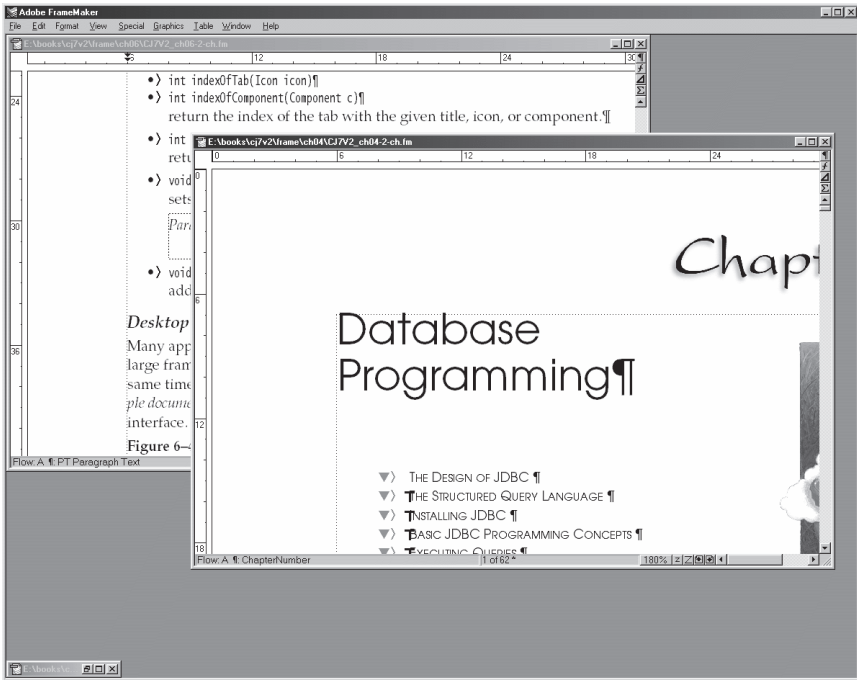


Рис. 6.47. Приложение с многодокументным интерфейсом

На рис. 6.48 показано приложение Java с тремя внутренними фреймами. Два из них содержат пиктограммы для полного развертывания, свертывания и закрытия фрейма. А третий фрейм показан в полностью свернутом виде.

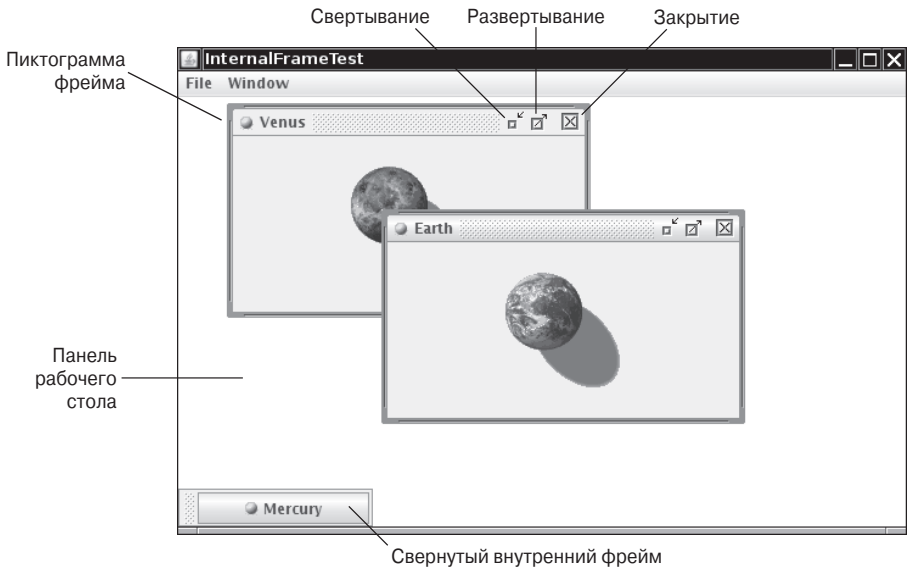


Рис. 6.48. Окно Java-приложения с тремя внутренними фреймами

При использовании стиля Metal внутренние фреймы имеют области “захвата”, которые можно использовать для перемещения фреймов. А для изменения размеров окон следует просто перетащить в нужном направлении уголки фрейма.

Для работы с фреймами выполните перечисленные ниже действия.

1. Создайте для приложения класс `JFrame`.
2. Добавьте к `JFrame` панель рабочего стола `JDesktopPane`:

```
desktop = new JDesktopPane();
setContentPane(desktop);
```

3. Создайте внутренние фреймы `JInternalFrame`. При этом можно также сформировать пиктограммы для изменения размеров фрейма и его закрытия:

```
JInternalFrame iframe =
  new JInternalFrame(title,
    true,  // можно изменять размер
    true,  // можно закрывать
    true,  // можно полностью развертывать
    true); // можно свертывать
```

4. Включите в состав фрейма документы:

```
iframe.add(c, BorderLayout.CENTER);
```

5. Создайте пиктограмму фрейма, которая будет представлена в его левом верхнем углу:

```
iframe.setFrameIcon(icon);
```



В текущей версии стиля Metal, когда фрейм находится в полностью свернутом виде, пиктограмма не отображается.

6. Установите размер внутреннего фрейма. В исходном состоянии внутренний фрейм, как и стандартный фрейм, имеет размеры 0×0. Кроме того, чтобы фреймы не закрывали друг друга, нужно указать разные позиции для всех фреймов с помощью метода `reshape()`. Этот же метод дает возможность установить размеры фреймов.

```
iframe.reshape(nextFrameX, nextFrameY, width, height);
```

7. Далее сделайте фрейм видимым:

```
iframe.setVisible(true);
```



В ранних версиях Swing внутренние фреймы автоматически становились видимыми, и этот метод вызывать не требовалось.

8. Включите фрейм в состав `JDesktopPane`:

```
desktop.add(iframe);
```

9. Вероятно, новый фрейм потребуется сделать *выделенным фреймом*, который, в отличие от других внутренних фреймов, имеет фокус ввода. Согласно стилю Metal строка заголовка выбранного фрейма имеет синий цвет, тогда как строка заголовка других невыбранных фреймов серая. Для выбора фрейма используется метод `setSelected()`. Кроме того, на выбранный фрейм можно наложить *запрет* (veto), т.е. текущий выбранный фрейм может отказать в передаче фокуса ввода. В этом случае метод `setSelected()` генерирует исключение `PropertyVetoException`, которое нужно обработать.

```
try
{
   iframe.setSelected(true);
}
catch(PropertyVetoException e)
{
   // наложен запрет на передачу фокуса
}
```

10. Чтобы фреймы не закрывали полностью друг друга, каждый последующий фрейм необходимо располагать со смещением. Удобно выбрать размер смещения равным высоте строки заголовка фрейма:

```
int frameDistance =
   iframe.getHeight() - iframe.getContentPane().getHeight();
```

11. Вычисленное расстояние используется для определения расположения следующего внутреннего фрейма:

```
nextFrameX += frameDistance;
nextFrameY += frameDistance;
if(nextFrameX + width > desktop.getWidth())
   nextFrameX = 0;
if(nextFrameY + width > desktop.getWidth())
   nextFrameY = 0;
```

Каскадное и мозаичное расположение фреймов

В операционной системе Windows предусмотрено несколько команд для организации *каскадного* (cascading) расположения фреймов с перекрытием (рис. 6.49) и *мозаичного* (tiling) расположения фреймов без перекрытия (рис. 6.50). Однако в классах `JDesktopPane` и `JInternalFrame` не предусмотрено никаких средств для поддержки этих операций. В листинге 6.21 показано, как можно самостоятельно организовать такое упорядочение фреймов.

Для каскадного расположения нужно указать для фреймов одинаковый размер и указывать их позиции с одинаковым шагом.

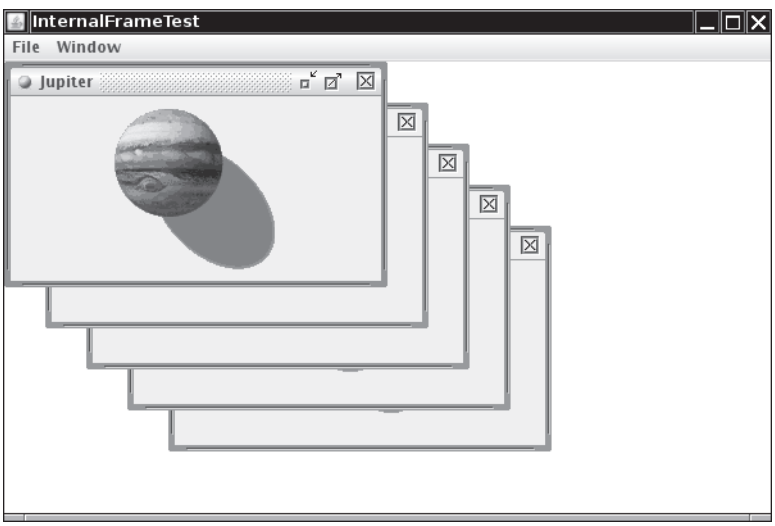


Рис. 6.49. Каскадное расположение фреймов

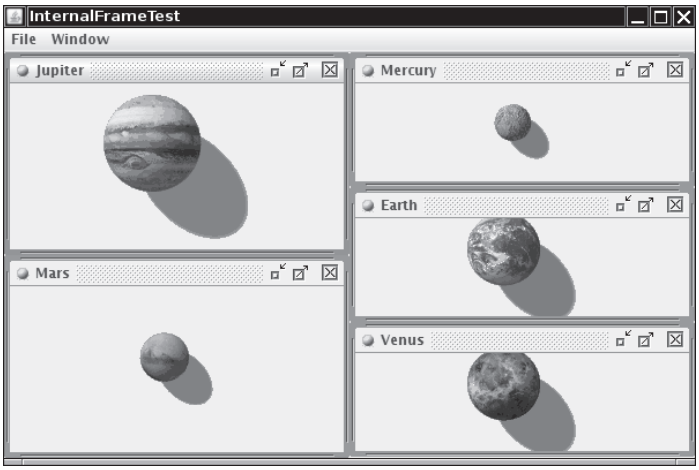


Рис. 6.50. Мозаичное расположение фреймов

Для получения массива всех внутренних фреймов используется метод `getAllFrames()` класса `JDesktopPane`:

```
JInternalFrame[] frame = desktop.getAllFrames();
```

Однако при этом нужно учитывать текущее состояние фрейма, которое может быть таким:

- пиктограмма, т.е. фрейм полностью свернут;
- фрейм имеет обычные размеры и допускает их изменение;
- фрейм полностью развернут.

Для поиска полностью свернутых фреймов, которые нужно пропустить, следует применять метод `isIcon()`. Для полностью развернутых фреймов нужно задать их обычный размер с помощью метода `setMaximum(false)`. На изменение этого свойства также может быть наложен запрет, следовательно, необходимо организовать перехват исключения `PropertyVetoException`.

Ниже представлен цикл перебора всех внутренних фреймов панели рабочего стола.

```
for (JInternalFrame frame : desktop.getAllFrames())
{
  if (!frame.isIcon())
  {
    try
    {
      // Попытка максимального развертывания фрейма, что может быть запрещено.
      frame.setMaximum(false);
      frame.reshape(x, y, width, height);
      x += frameDistance;
      y += frameDistance;
      // Окна достигли краев рабочего стола.
      if (x + width > desktop.getWidth()) x = 0;
      if (y + height > desktop.getHeight()) y = 0;
    }
    catch (PropertyVetoException e)
    {}
  }
}
```

Мозаичное расположение фреймов организовать гораздо сложнее, особенно если количество фреймов таково, что они не образуют в точности квадратную решетку. Сначала необходимо подсчитать количество несвернутых фреймов, а затем определить количество строк мозаики:

```
int rows = (int) Math.sqrt(frameCount);
```

После этого определяется количество столбцов за исключением последнего:

```
int cols = frameCount / rows;
```

Последний незаполненный столбец выявляется так:

```
int extra = frameCount % rows;
```

Ниже представлен цикл перебора всех внутренних фреймов рабочего стола, которые будут расположены в виде мозаики:

```
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)
        {
          // Добавление дополнительной строки.
          rows++;
          height = desktop.getHeight() / rows;
         }
      }
    }
    catch (PropertyVetoException e)
    {}
  }
}
```

В программе, которая будет рассмотрена далее, показана еще одна распространенная операция: перемещение выбора от текущего фрейма к следующему несвернутому фрейму. Нужно вручную организовать обход всех фреймов и вызов метода `isSelected()` до тех пор, пока не будет найден текущий выбранный фрейм. Затем нужно найти следующий несвернутый фрейм и попытаться выбрать его с помощью приведенного ниже метода:

```
frames[next].setSelected(true);
```

Как и прежде, этот метод может генерировать исключение `PropertyVetoException`. Возврат к исходному фрейму означает, что не существует других доступных для выбора фреймов. Покажем, как выглядит цикл перебора всех внутренних фреймов.

```
JInternalFrame[] frames = desktop.getAllFrames();
for (int i = 0; i < frames.length; i++)
{
   if (frames[i].isSelected())
   {
      // Поиск следующего несвернутого фрейма,
      // который может быть выбран.
      int next = (i + 1) % frames.length;
      while (next != i)
      {
         if (!frames[next].isIcon())
         {
            try
            {
               // Все остальные фреймы свернуты, либо выбирать их запрещено.
               frames[next].setSelected(true);
               frames[next].toFront();
               frames[i].toBack();
               return;
            }
            catch (PropertyVetoException e)
            {}
         }
         next = (next + 1) % frames.length;
      }
   }
}
```

Запрет на изменение свойства

Рассмотрим способы генерирования исключений в случае наложения запрета на изменение свойства. В классе `JInternalFrame` для слежения за значениями свойств используется механизм *JavaBeans*. Этот механизм подробно описывается в главе 8, а здесь мы кратко рассмотрим только тот материал, который относится к обработке запретов на изменение свойства.

Запреты обычно накладываются не на свертывание или передачу фокуса ввода, а на *закрытие* фрейма. Для закрытия фрейма используется метод `setClosed()` класса `JInternalFrame`. Поскольку для этого метода предусмотрено использование запретов, то перед изменением состояния фрейма он вызывает все зарегистрированные *обработчики запрещаемых изменений* (vetoable change listeners).

Таким образом, каждому обработчику предоставляется возможность сгенерировать исключение `PropertyVetoException` и завершить метод `setClosed()` до выполнения им каких-либо действий.

В программе, рассматриваемой в качестве примера, отображается диалоговое окно, показанное на рис. 6.51. В нем пользователю предлагается подтвердить намерение закрыть фрейм. Если пользователь не дает такого подтверждения, то фрейм останется открытым.

Для организации такого поведения выполните перечисленные ниже действия.

1. Создайте обработчик для каждого фрейма. Это должен быть экземпляр класса, реализующего интерфейс `VetoableChangeListener`. Обработчик лучше всего связать с фреймом сразу же после создания последнего. В данном примере для создания внутренних фреймов применяется класс фрейма, хотя для этого можно также использовать безымянный внутренний класс:

   ```
   iframe.addVetoableChangeListener(listener);
   ```

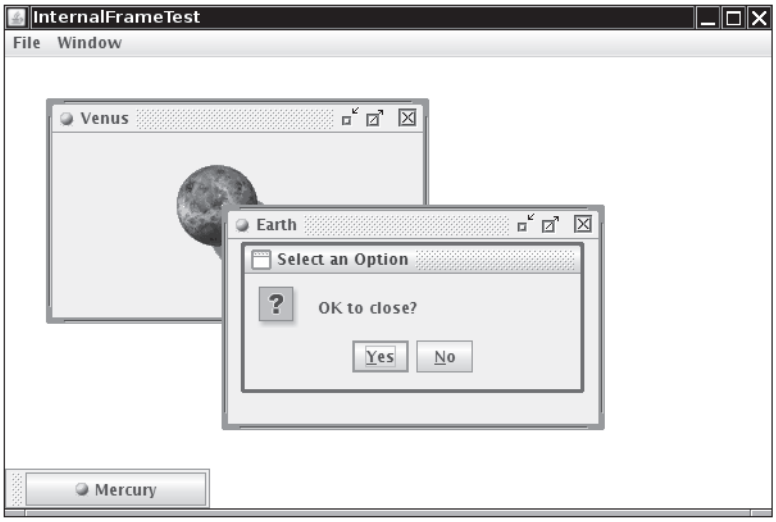


Рис. 6.51. Обработка запрета на закрытие окна

2. Реализуйте метод `vetoableChange()` — единственный, объявленный в интерфейсе `VetoableChangeListener`. Ему передается параметр типа `PropertyChangeEvent`. Для получения имени изменяемого свойства воспользуйтесь методом `getName()`, например, если запрет задается для метода `setClosed(true)`, то именем свойства будет `closed`. Как показано в главе 8, имя свойства получается за счет удаления префикса `set` из имени метода и перевода первой буквы в нижний регистр. Для получения нового значения свойства следует использовать метод `getNewValue()`:

```
String name = event.getPropertyName();
Object value = event.getNewValue();
if (name.equals("closed") && value.equals(true))
{
   // Предложение пользователю подтвердить закрытие фрейма.
}
```

3. Генерируйте исключение `PropertyVetoException`, чтобы блокировать изменение свойства. Если запрет на изменение свойства не накладывается, метод `vetoableChange()` завершается обычным способом:

```
class DesktopFrame extends JFrame
   implements VetoableChangeListener
{
   . . .
   public void vetoableChange(PropertyChangeEvent event)
      throws PropertyVetoException
   {
      . . .
      if (фрейм не должен быть закрыт)
         throw new PropertyVetoException(reason, event);
      // Если фрейм может быть закрыт, обычный возврат.
   }
}
```

Диалоговые окна во внутренних фреймах

Для создания диалоговых окон во внутренних фреймах не рекомендуется использовать класс `JDialog`, поскольку такие окна обладают следующими недостатками:

- они очень ресурсоемки, так как создают новый фрейм в системе окон;
- система управления окнами не знает, как расположить диалоговое окно по отношению к породившему его внутреннему фрейму.

Вместо этого для создания простых диалоговых окон рекомендуется использовать методы `showInternal`*`Xxx`*`Dialog()` класса `JOptionPane`. Они работают аналогично методам `show`*`Xxx`*`Dialog()`, 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 поддержки модальных окон.

В рассматриваемом здесь примере используется внутреннее диалоговое окно, в котором пользователю предлагается подтвердить закрытие фрейма:

```
int result = JOptionPane.showInternalConfirmDialog(iframe,
  "OK to close?", "Select an Option", JOptionPane.YES_NO_OPTION);;
```



Если необходимо лишь *уведомление*, можно не использовать механизм запрета и применять обработчик `InternalFrameListener`, который работает аналогично обработчику `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 диспетчер панели постоянно перерисовывает фрейм, что существенно замедляет работу.

Такая ситуация наблюдается при работе в графических оболочках Windows или X Window с неудачно написанным видеодрайвером. В некоторых системах перетаскивание выполняется быстро, так как видеокарты поддерживают данную операцию на аппаратном уровне.

Для повышения производительности можно использовать метод перетаскивания контуров. В таком случае при перетаскивании фрейма постоянно обновляются только его контуры. Внутренняя часть будет перерисована, только когда фрейм достигнет целевой позиции.

Для включения режима перетаскивания контуров нужно вызвать следующий метод:

```
desktop.setDragMode(JDesktopPane.OUTLINE_DRAG_MODE);
```

Эта установка эквивалентна переключению режима постоянной перерисовки в классе `JSplitPane`.



В ранних версиях Swing для включения перетаскивания контуров нужно было использовать метод:

```
desktop.putClientProperty("JDesktopPane.dragMode", "outline");
```

В данном примере для включения/отключения режима перетаскивания контуров пользователю предоставляется пункт меню Window⇨Drag Outline (Окно⇨Перетащить контур).



Внутренними фреймами панели рабочего стола управляет диспетчер `DesktopManager`. При использовании стандартных приемов программирования этот класс не потребуется. Он может применяться для реализации разных вариантов поведения панели, но в данной книге эти вопросы не рассматриваются.

В листинге 6.21 приведен пример панели рабочего стола с внутренними фреймами, предназначенными для отображения HTML-страниц. При активизации пункта меню File⇨Open (Файл⇨Открыть) на экране отображается диалоговое окно для выбора локального HTML-файла. После щелчка на выбранном HTML-файле этот файл будет отображен на экране 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 6.21. Содержимое файла `InternalFrameTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.beans.*;
4.  import javax.swing.*;
5.
6.  /**
7.   * Данная программа демонстрирует использование
8.   * внутренних фреймов.
9.   * @version 1.11 2007-08-01
10.  * @author Cay Horstmann
11.  */
12. public class InternalFrameTest
13. {
14.    public static void main(String[] args)
15.    {
16.       EventQueue.invokeLater(new Runnable()
17.          {
18.             public void run()
19.             {
20.                JFrame frame = new DesktopFrame();
21.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22.                frame.setVisible(true);
23.             }
24.          });
25.    }
26. }
27.
28. /**
29.  * Фрейм содержит панели редактирования,
30.  * в которых отображаются HTML-документы.
31.  */
```

```
32.  class DesktopFrame extends JFrame
33.  {
34.     public DesktopFrame()
35.     {
36.        setTitle("InternalFrameTest");
37.        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
38.
39.        desktop = new JDesktopPane();
40.        add(desktop, BorderLayout.CENTER);
41.
42.        // Создание меню.
43.
44.        JMenuBar menuBar = new JMenuBar();
45.        setJMenuBar(menuBar);
46.        JMenu fileMenu = new JMenu("File");
47.        menuBar.add(fileMenu);
48.        JMenuItem openItem = new JMenuItem("New");
49.        openItem.addActionListener(new ActionListener()
50.           {
51.              public void actionPerformed(ActionEvent event)
52.              {
53.                 createInternalFrame(
54.                    new JLabel(new ImageIcon(planets[counter] + ".gif")),
55.                               planets[counter]);
56.                 counter = (counter + 1) % planets.length;
57.              }
58.           });
59.        fileMenu.add(openItem);
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.        JMenu windowMenu = new JMenu("Window");
70.        menuBar.add(windowMenu);
71.        JMenuItem nextItem = new JMenuItem("Next");
72.        nextItem.addActionListener(new ActionListener()
73.           {
74.              public void actionPerformed(ActionEvent event)
75.              {
76.                 selectNextWindow();
77.              }
78.           });
79.        windowMenu.add(nextItem);
80.        JMenuItem cascadeItem = new JMenuItem("Cascade");
81.        cascadeItem.addActionListener(new ActionListener()
82.           {
83.              public void actionPerformed(ActionEvent event)
84.              {
85.                 cascadeWindows();
86.              }
87.           });
```

```
88.       windowMenu.add(cascadeItem);
89.       JMenuItem tileItem = new JMenuItem("Tile");
90.       tileItem.addActionListener(new ActionListener()
91.          {
92.             public void actionPerformed(ActionEvent event)
93.             {
94.                tileWindows();
95.             }
96.          });
97.       windowMenu.add(tileItem);
98.       final JCheckBoxMenuItem dragOutlineItem =
99.          new JCheckBoxMenuItem("Drag Outline");
100.      dragOutlineItem.addActionListener(new ActionListener()
101.         {
102.            public void actionPerformed(ActionEvent event)
103.            {
104.               desktop.setDragMode(dragOutlineItem.isSelected() ?
105.                  JDesktopPane.OUTLINE_DRAG_MODE : JDesktopPane.LIVE_DRAG_MODE);
106.            }
107.         });
108.      windowMenu.add(dragOutlineItem);
109.   }
110.
111.   /**
112.    * Создание внутреннего фрейма в панели рабочего стола.
113.    * @param c Компонент для отображения во внутреннем фрейме
114.    * @param t Заголовок внутреннего фрейма
115.    */
116.   public void createInternalFrame(Component c, String t)
117.   {
118.      final JInternalFrame iframe = new JInternalFrame(t, true,
119.            // Допускается изменение размеров.
120.         true, // Допускается закрытие.
121.         true, // Допускается максимизация.
122.         true); // Допускается свертывание.
123.
124.      iframe.add(c, BorderLayout.CENTER);
125.      desktop.add(iframe);
126.
127.      iframe.setFrameIcon(new ImageIcon("document.gif"));
128.
129.      // Обработчик для подтверждения закрытия фрейма.
130.      iframe.addVetoableChangeListener(new VetoableChangeListener()
131.         {
132.            public void vetoableChange(PropertyChangeEvent event)
133.               throws PropertyVetoException
134.            {
135.               String name = event.getPropertyName();
136.               Object value = event.getNewValue();
137.
138.               // Нам необходимо лишь следить за попытками
139.               // закрыть фрейм.
140.               if (name.equals("closed") && value.equals(true))
141.               {
142.                  // Запрос подтверждения у пользователя.
143.                  int result =
144.                     JOptionPane.showInternalConfirmDialog( iframe, "OK to close?",
145.                        "Select an Option JOptionPane.YES_NO_OPTION);
```

```
146.                // Если пользователь не подтвердил свои
147.                // намерения, фрейм не закрывается.
148.                if (result != JOptionPane.YES_OPTION)
149.                  throw new PropertyVetoException("User canceled close", event);
150.              }
151.            }
152.          });
153.
154.      // Позиционирование фрейма.
155.      int width = desktop.getWidth() / 2;
156.      int height = desktop.getHeight() / 2;
157.      iframe.reshape(nextFrameX, nextFrameY, width, height);
158.
159.      iframe.show();
160.
161.      // Выбор фрейма - операция может быть запрещена.
162.      try
163.      {
164.        iframe.setSelected(true);
165.      }
166.      catch (PropertyVetoException e)
167.      {}
168.
169.      frameDistance = iframe.getHeight() -
170.        iframe.getContentPane().getHeight();
171.
172.      // Вычисление позиции следующего фрейма.
173.
174.      nextFrameX += frameDistance;
175.      nextFrameY += frameDistance;
176.      if (nextFrameX + width > desktop.getWidth()) nextFrameX = 0;
177.      if (nextFrameY + height > desktop.getHeight()) nextFrameY = 0;
178.    }
179.    /**
180.     * Каскадное расположение несвернутых внутренних фреймов.
181.     */
182.    public void cascadeWindows()
183.    {
184.      int x = 0;
185.      int y = 0;
186.      int width = desktop.getWidth() / 2;
187.      int height = desktop.getHeight() / 2;
188.
189.      for (JInternalFrame frame : desktop.getAllFrames())
190.      {
191.        if (!frame.isIcon())
192.        {
193.          try
194.          {
195.            // Попытка установить нормальные размеры для
196.            // максимизированного фрейма. Операция может
197.            // быть запрещена.
198.            frame.setMaximum(false);
199.            frame.reshape(x, y, width, height);
200.
201.            x += frameDistance;
202.            y += frameDistance;
```

```
            // Достижение фреймами границ окна.
            if (x + width > desktop.getWidth()) x = 0;
            if (y + height > desktop.getHeight()) y = 0;
         }
         catch (PropertyVetoException e)
         {
         }
      }
   }
}
/**
 * Мозаичное расположение несвернутых внутренних фреймов.
 */
public void tileWindows()
{
   // Подсчет несвернутых фреймов.
   int frameCount = 0;
   for (JInternalFrame frame : desktop.getAllFrames())
      if (!frame.isIcon()) frameCount++;
   if (frameCount == 0) return;

   int rows = (int) Math.sqrt(frameCount);
   int cols = frameCount / rows;
   int extra = frameCount % rows;
   // Число столбцов в незаполненной строке.

   int width = desktop.getWidth() / Столбцы;
   int height = desktop.getHeight() / Строки;
   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)
               {
                  // Добавление дополнительной строки.
                  rows++;
                  height = desktop.getHeight() / Строки;
               }
            }
         }
         catch (PropertyVetoException e)
         {}
      }
   }
}
```

```
260.    /**
261.     * Перевод следующего несвернутого фрейма на передний план.
262.     */
263.    public void selectNextWindow()
264.    {
265.       JInternalFrame[] frames = desktop.getAllFrames();
266.       for (int i = 0; i < frames.length; i++)
267.       {
268.          if (frames[i].isSelected())
269.          {
270.             // Поиск следующего несвернутого фрейма, который может быть выбран.
271.             int next = (i + 1) % frames.length;
272.             while (next != i)
273.             {
274.                if (!frames[next].isIcon())
275.                {
276.                   try
277.                   {
278.                      // Остальные фреймы либо свернуты,
279.                      // либо их выбор запрещен.
280.                      frames[next].setSelected(true);
281.                      frames[next].toFront();
282.                      frames[i].toBack();
283.                      return;
284.                   }
285.                   catch (PropertyVetoException e)
286.                   {
287.                   }
288.                }
289.                next = (next + 1) % frames.length;
290.             }
291.          }
292.       }
293.    }
294.    private JDesktopPane desktop;
295.    private int nextFrameX;
296.    private int nextFrameY;
297.    private int frameDistance;
298.    private int counter;
299.    private static final String[] planets = { "Mercury", "Venus", "Earth",
300.           "Mars", "Jupiter", "Saturn", "Uranus", "Neptune", "Pluto", };
301.    private static final int DEFAULT_WIDTH = 600;
302.    private static final int DEFAULT_HEIGHT = 400;
303. }
```

API javax.swing.JDesktopPane 1.2

- `JInternalFrame[] getAllFrames()`

  Возвращает все внутренние фреймы данной панели.

- `void setDragMode(int mode)`

  Задает на время перетаскивания фрейма отображение его содержимого или только контуров.

  Параметр: `mode` Одно из значений: `JDesktopPane.LIVE_DRAG_MODE` и `JDesktopPane.OUTLINE_DRAG_MODE`

API 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)`

Создает новый внутренний фрейм.

Параметры:		
	`title`	Заголовок фрейма
	`resizable`	Значение `true` допускает изменение размеров фрейма
	`closeable`	Значение `true` допускает закрытие фрейма
	`maximazable`	Значение `true` допускает развертывание фрейма
	`iconifiable`	Значение `true` допускает свертывание 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)`

Возвращает или устанавливает свойства `resizable`, `closable`, `maximizable` и `iconifiable`. Если соответствующее свойство равно `true`, в строке заголовка фрейма отображается пиктограмма изменения размеров, закрытия, минимизации или свертывания внутреннего фрейма.

- `boolean isIcon()`
- `void setIcon(boolean b)`
- `boolean isMaximum()`
- `void setMaximum(boolean b)`
- `boolean isClosed()`
- `void setClosed(boolean b)`

Возвращает значение свойства `icon`, `maximum` или `closed`. Если значение соответствующего свойства равно `true`, то допускается свертывание, максимизация или закрытие внутреннего фрейма.

- `boolean isSelected()`
- `void setSelected(boolean b)`

Возвращает или устанавливает значение свойства `selected`. Если данное свойство имеет значение `true`, то текущий внутренний фрейм становится выбранным.

- `void moveToFront()`
- `void moveToBack()`

Перемещает внутренний фрейм на передний план или отменяет отображение на переднем плане.

- `void reshape(int x, int y, int width, int height)`
  Перемещает внутренний фрейм и изменяет его размеры.

Параметры:	`x, y`	Координаты верхнего левого угла фрейма
	`width, height`	Ширина и высота фрейма

- `Container getContentPane()`
- `void setContentPane(Container c)`
  Возвращает или устанавливает панель содержимого для данного внутреннего фрейма.
- `JDesktopPane getDesktopPane()`
  Возвращает панель рабочего стола для данного внутреннего фрейма.
- `Icon getFrameIcon()`
- `void setFrameIcon(Icon anIcon)`
  Возвращает или устанавливает пиктограмму фрейма, которая отображается в строке заголовка.
- `boolean isVisible()`
- `void setVisible(boolean b)`
  Возвращает или устанавливает значение свойства, определяющего, должен ли фрейм быть видимым.
- `void show()`
  Отображает внутренний фрейм и перемещает его на передний план.

API javax.swing.JComponent 1.2

- `void addVetoableChangeListener(VetoableChangeListener listener)`
  Создает обработчик запрещаемых изменений, который уведомляется при попытке изменения свойства.

API java.beans.VetoableChangeListener 1.1

- `void vetoableChange(PropertyChangeEvent event)`
  Вызывается, когда метод `set()` свойства с наложенным на него ограничением уведомляет обработчик изменений.

API java.beans.PropertyChangeEvent 1.1

- `String getPropertyName()`
  Возвращает имя изменяемого свойства.
- `Object getNewValue()`
  Возвращает предлагаемое новое значение свойства.

API java.beans.PropertyVetoException 1.1

- `PropertyVetoException(String reason, PropertyChangeEvent event)`
  Генерирует исключение для свойства, изменения которого запрещены.

Параметры:	`reason`	Причина запрета
	`event`	Запрещенное событие

Теперь вы знаете, как используются сложные компоненты, предлагаемые библиотекой Swing. В следующей главе мы займемся рассмотрением вопросов, связанных с AWT: сложные операции рисования, манипуляция изображениями, печать и организация интерфейсов с оконной системой.

Глава 7

РАСШИРЕННЫЕ СРЕДСТВА AWT

В ЭТОЙ ГЛАВЕ...

- Алгоритм рисования
- Фигуры
- Области
- Штрихи
- Расцветки
- Преобразование координат
- Отсечение
- Прозрачность и композиция
- Параметры рисования
- Чтение и запись изображений
- Управление изображениями
- Печать
- Буфер обмена
- Копирование путем перетаскивания
- Средства интеграции в платформу

Для создания простых рисунков можно использовать методы класса Graphics. Они эффективны при работе над простыми аплетами и приложениями, но их недостаточно для построения сложных фигур или для осуществления полного контроля над внешним видом рисунка. Java 2D API содержит самую последнюю версию библиотеки классов, которую можно использовать для создания высококачественных рисунков. В данной главе рассматриваются основные способы применения этой библиотеки. Кроме того, в ней описываются вопросы управления печатью и обсуждаются способы печати, которые можно реализовать в вашей программе.

В конце главы рассматриваются две технологии обмена данными между программами: с помощью буфера обмена и механизма перетаскивания. Эти методы можно использовать для передачи информации между двумя приложениями Java или между приложением Java и другими программами, в том числе платформенно-ориентированными.

Алгоритм рисования

В исходной версии пакета JDK 1.0 (Java Development Kit) механизм для рисования фигур выглядел очень просто. Можно было только выбирать необходимый цвет и режим рисования и вызывать методы класса `Graphics`, вроде `drawRect()` или `fillOval()`. API-интерфейс в Java 2D поддерживает гораздо больше возможностей. В частности он позволяет делать следующее:

- легко создавать множество различных *фигур*;
- управлять *штрихом*, то есть пером, прорисовывающим границы фигур;
- *заливать* фигуры любым сплошным цветом, используя различные оттенки и узоры;
- использовать *трансформации* для перемещения, масштабирования, поворачивания и растягивания фигур;
- *отсекать* фигуры так, чтобы они занимали только какую-то определенную область;
- выбирать *правила композиции* для описания того, как пиксели новой фигуры должны комбинироваться с пикселями уже существующей фигуры;
- задавать *рекомендации по визуализации* для достижения компромисса между скоростью загрузки и качеством рисования.

Для рисования фигуры требуется выполнять следующие действия.

1. Получать объект класса `Graphics2D`. Этот класс является подклассом класса `Graphics`. Начиная с версии Java SE 1.2, методы вроде `paint()` и `paintComponent()` автоматически получают объект класса `Graphics2D`. Поэтому остается лишь применить приведение типа, как показано ниже:

```
public void paintComponent(Graphics g)
{
   Graphics2D g2 = (Graphics2D)g;
}
```

2. Использовать метод `setRenderingHints()` для добавления *рекомендаций по визуализации* (rendering hints) в целях достижения компромисса между скоростью и качеством рисования.

```
RenderingHints hints = ...;
g2. SetRenderingHints(hints);
```

3. Использовать метод `setStroke()` для указания *штриха* (stroke), который должен применяться для прорисовки контура фигуры. Для этого штриха можно выбирать толщину, а также сплошную или пунктирную линию.

```
Stroke stroke = ...;
g2.setStroke(stroke);
```

4. Использовать метод `setPaint()` для указания *расцветки* (paint). Расцветка подразумевает закрашивание областей, вроде пути штриха или внутренней части фигуры. Она может состоять из одного сплошного цвета, нескольких меняющихся оттенков или мозаичных узоров.

```
Paint paint = ...;
g2.setPaint(paint);
```

5. Использовать метод `setClip()` для установки *области отсечения*.

```
Shape clip = ...;
g2.clip(clip);
```

6. Использовать метод `setTransform()` для *преобразования* (трансформации) координат рисунка из относительной пользовательской системы координат в абсолютную систему координат устройства. Это преобразование следует применять в тех случаях, когда проще создать фигуру в пользовательской системе координат, чем использовать координаты, выраженные в пикселях.

```
AffineTransform transform = ...;
g2.transform(transform);
```

7. Использовать метод `setComposite()` для задания *правил композиции*, описывающих, каким именно образом новые пиксели должны объединяться с уже существующими.

```
Composite (composite) = ...;
g2.setComposite(composite);
```

8. Создавать фигуру. В Java 2D API предусмотрено много объектов-фигур и методов для их комбинирования.

```
Shape shape = ...;
```

9. Рисовать или заливать фигуру. Под рисованием подразумевается очерчивание контуров фигуры, а под заливкой — закрашивание ее внутренней части.

```
g2.draw(shape);
g2.fill(shape);
```

Разумеется, на практике выполнять все эти действия обычно не нужно, потому для многих параметров двухмерного (2D) графического контекста предусмотрены значения по умолчанию, изменять которые можно только в случае особой необходимости.

В следующем разделе рассматриваются способы описания фигур, штрихов, расцветок, преобразований и правил композиции.

Различные `set`-методы устанавливают состояние двухмерного графического контекста и не относятся к конкретным рисункам. Например, изображение генерируется не во время создания объекта `Shape`, а при вызове методов `draw()` или `fill()` на основании показанного на рис. 7.1 *алгоритма рисования* (rendering pipeline).

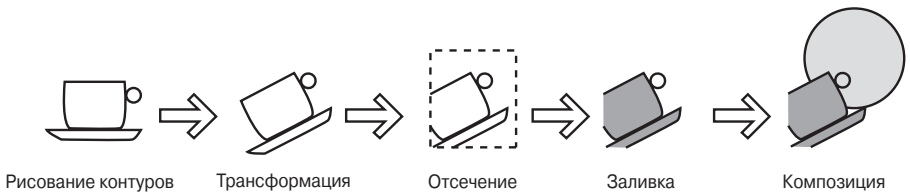


Рис. 7.1. Алгоритм рисования

Для рисования изображения в соответствии с этим алгоритмом выполняются перечисленные ниже действия.

1. Очерчиваются контуры фигуры.
2. Фигура преобразуется.
3. Если фигура выходит за пределы области отсечения, фрагменты фигуры удаляются.
4. Оставшаяся после отсечения часть фигуры заполняется в соответствии с выбранной расцветкой.
5. Новая фигура объединяется с существующей. Это видно из рис. 7.1, где круг представляет существующую фигуру, а чашка накладывается поверх него.

В следующем разделе рассматриваются способы определения фигур, а затем описываются установки двухмерного графического контекста.

API java.awt.Graphics2D 1.2

- `void draw(Shape s)`

  Рисует контуры данной фигуры с текущими параметрами штриха.

- `void fill(Shape s)`

  Заполняет внутреннюю часть фигуры в соответствии с текущей расцветкой.

Фигуры

Ниже перечислено несколько методов класса `Graphics`, которые используются для рисования фигур:

```
drawLine
drawRectangle
drawRoundRect
draw3DRect
drawPolygon
drawPolyline
drawOval
drawArc
```

Для них существуют соответствующие `fill`-методы заполнения фигур, которые были предусмотрены в классе `Graphics` начиная с JDK 1.0. В Java 2D API используется совершенно другая, объектно-ориентированная модель, где вместо методов применяются перечисленные ниже классы:

```
Line2D
Rectangle2D
RoundRectangle2D
Ellipse2D
Arc2D
QuadCurve2D
CubicCurve2D
GeneralPath
```

Данные классы реализуют интерфейс `Shape`.

Кроме того, предусмотрен класс точки Point2D, который описывает точку с координатами x и y. Точки полезны для определения фигур, но сами по себе фигурами не являются.

Для того чтобы нарисовать фигуру, нужно создать экземпляр класса, который реализует интерфейс Shape, а затем вызвать метод draw для класса Graphics2D.

Классы Line2D, Rectangle2D, RoundRectangle2D, Ellipse2D и Arc2D соответствуют методам drawLine(), drawRectangle(), drawRoundRect(), drawOval() и drawArc(). Для понятия "трехмерный прямоугольник" (3D rectangle) не предусмотрено соответствующего метода draw3DRect(). Однако в Java 2D API поддерживаются два дополнительных класса для рисования кривых второго и третьего порядков. Эти фигуры будут рассмотрены далее в главе. Кроме того, для рисования многоугольников не предусмотрено отдельного класса (вроде Polygon2D), а предлагается класс GeneralPath, который описывает контуры, состоящие из линий и кривых второго и третьего порядков. Класс GeneralPath можно использовать для описания многоугольника, которое обсуждается далее в этом разделе.

Перечисленные ниже классы наследуют общий класс RectangularShape:

```
Rectangle2D
RoundRectangle2D
Ellipse2D
Arc2D
```

Известно, что эллипсы и дуги не являются прямоугольниками, но их можно вписать в прямоугольник (рис. 7.2).

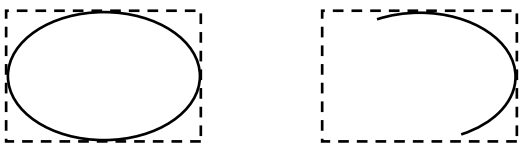


Рис. 7.2. Эллипс и дугу можно вписать в прямоугольник

Каждый из классов, название которого заканчивается суффиксом 2D, имеет два подкласса, отличающиеся способом указания координат (они задаются в виде чисел типа float и double). Например, в первом томе уже встречались выражения Rectangle2D.Float и Rectangle2D.Double.

Такая же схема используется и для других классов, например Arc2D.Float и Arc2D.Double.

Для внутреннего представления координат все графические классы используют данные типа float, поскольку для них требуется меньше места для хранения, чем для данных типа double, к тому же они поддерживают достаточную точность для геометрических вычислений. Однако в языке Java обработка чисел float выполняется очень громоздкими и неуклюжими способами. Поэтому большинство методов графических классов используют параметры типа double и возвращают значения типа double. Выбирать тип данных (float или double) 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 *Xxx*2D.Float и *Xxx*2D.Double являются подклассами класса *Xxx*2D. После создания объекта нет надобности запоминать подклассы, а можно просто сохранить созданный объект в переменной суперкласса.

Судя по названиям *Xxx*2D.Float и *Xxx*2D.Double, можно предположить, что они являются внутренними классами классов *Xxx*2D. Это простое синтаксическое правило позволяет избежать увеличения длины имен внешних классов.

На рис. 7.3 показаны отношения между классами фигур без указания подклассов Double и Float. Традиционные классы библиотеки 2D, которые унаследованы из прежних версий JDK, закрашены серым цветом.

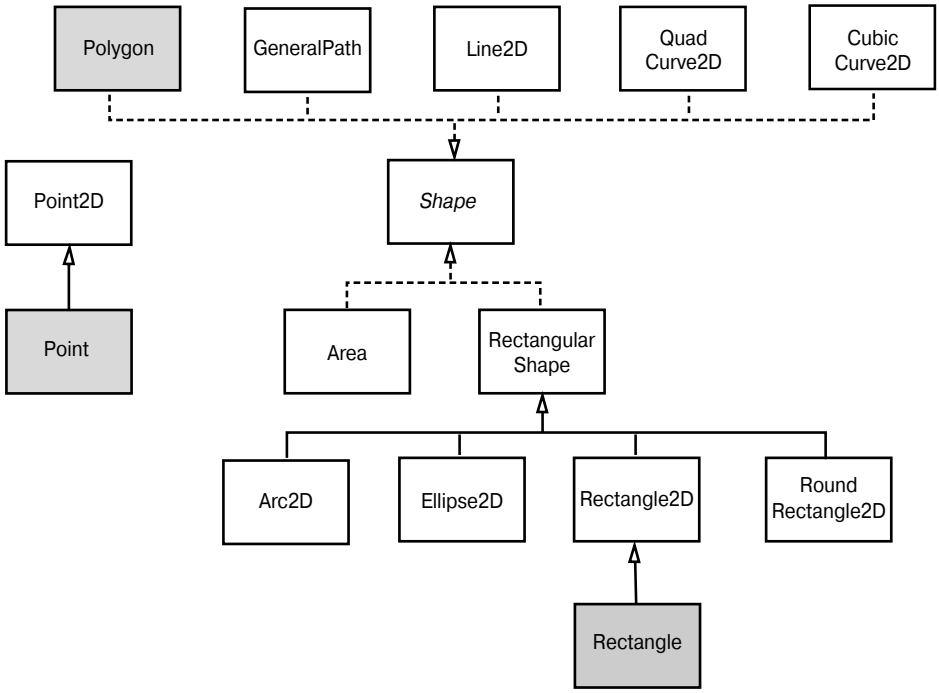


Рис. 7.3. Отношения между классами фигур

Классы фигур

Классы Line2D, Rectangle2D и Ellipse2D уже упоминались в главе 7 первого тома, а в этом разделе рассматриваются способы использования остальных двухмерных фигур.

Для фигуры RoundRectangle2D нужно указать координаты верхнего левого угла, ширину, высоту и параметры (*x* и *y*) области скругления углов (рис. 7.4). Например, в результате следующего вызова метода 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 дуги (рис. 7.5) и один из типов замыкания дуги (Arc2D.OPEN, Arc2D.PIE или Arc2D.CHORD):

```
Arc2D a = new Arc2D(x, y, width, height, startAngle, arcAngle, closureType);
```

На рис. 7.6 показаны типы замыкания дуги.

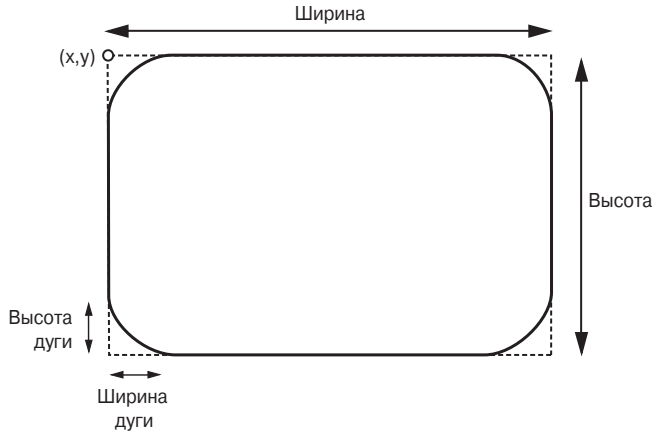


Рис. 7.4. Использование класса `RoundRectangle2D`

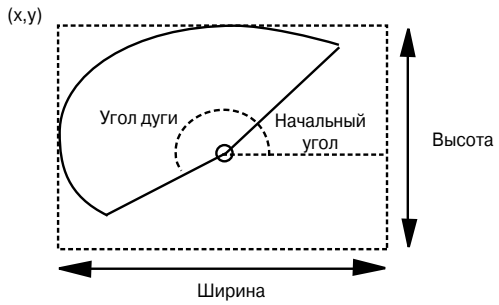


Рис. 7.5. Построение эллиптической дуги

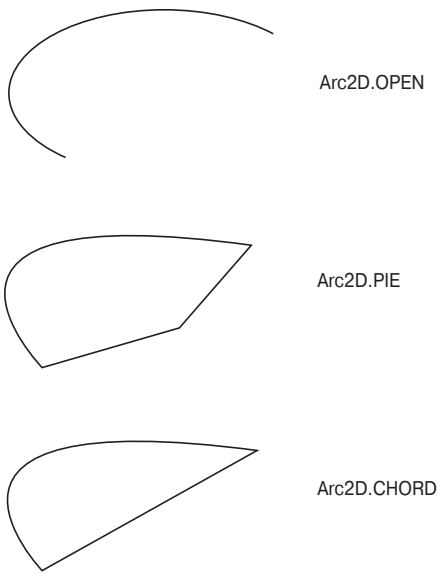


Рис. 7.6. Типы замыкания дуги



Если дуга имеет эллиптическую форму, процесс вычисления ее углов будет выглядеть далеко не просто. В документации по API сказано: “Углы задаются относительно неквадратного обрамляющего прямоугольника, так что 45 градусов всегда приходится на линию, выходящую из центра овала и оканчивающуюся в правом верхнем углу обрамляющего прямоугольника. Поэтому если обрамляющий прямоугольник является заметно длиннее по одной оси, чем по другой, углы к началу и концу сегмента дуги будут скашиваться вдоль более длинной оси фрейма”. К сожалению, там ничего не сказано о том, как вычислять это “скашивание”. Так вот, делается это следующим образом.

Предположим, что центр дуги приходится на начальную точку отсчета, а точка (*x, y*) находится на самой дуге. Формула для вычисления скошенного угла (skewed angle) выглядит следующим образом:

```
skewedAngle = Math.toDegrees(Math.atan2(x * width, y * height));
```

Результатом будет значение в диапазоне между –180 и 180. Вычислите скошенный начальный и конечный угол с помощью этой формулы, а затем вычислите разницу между этими двумя скошенными углами. Если получится негативное значение, прибавьте 360. Далее останется только предоставить значение начального угла и значение разницы между двумя углами в качестве параметров конструктору дуги.

Запустив демонстрационную программу, код которой будет приведен в конце этого раздела, вы сможете наглядно удостовериться в том, что такой подход действительно позволяет получать правильные значения для конструктора дуги (рис. 7.9).

В пакете Java 2D API предусмотрено использование кривых *второго* и *третьего* порядков. Здесь не будут рассматриваться математические аспекты создания этих кривых, но для демонстрации их возможностей предлагается запустить программу, код которой приведен в листинге 7.1. Как показано на рис. 7.7 и 7.8, кривые второго и третьего порядка определяются двумя *конечными точками* и одной *контрольной точкой* (для кривых третьего порядка контрольных точек две). Перемещение контрольных точек изменяет форму кривых.

Для создания квадратичных и кубических кривых задаются координаты конечных и контрольных точек:

```
QuadCurve2D q = new QuadCurve2D.Double(startX, startY, controlX,
   controlY, endX, endY);
CubicCurve2D c = new CubicCurve2D.Double(startX, startY,
   control1X, control1Y, control2X, control2Y, endX, endY);
```

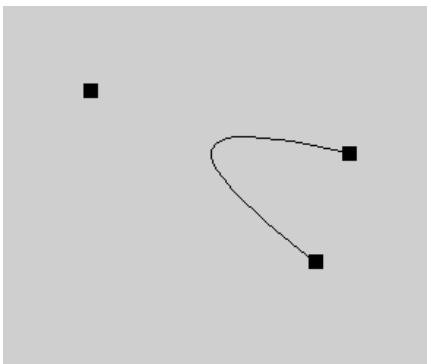


Рис. 7.7. Кривая второго порядка (квадратичная)

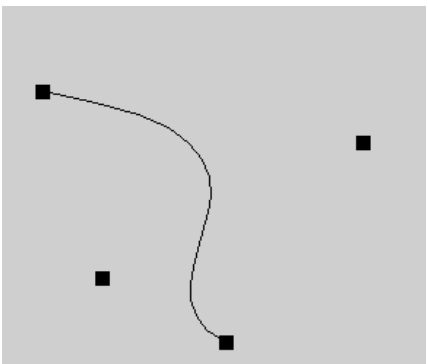


Рис. 7.8. Кривая третьего порядка (кубическая)

Кривые второго порядка не очень гибки и потому редко используются на практике, в отличие от кривых третьего порядка (например, кривая Безье, представляемая с помощью класса `CubicCurve2D`). Можно создавать сложные поверхности, комбинируя несколько кривых третьего порядка таким образом, чтобы совпадал их угол наклона в точках соединения. Более подробно об этом можно узнать из книги Джеймса Фоли (James D. Foley), Андриса ван Дама (Andries van Dam), Стивена Фэйнера (Steven K. Feiner) и др. *Computer Graphics: Principles and Practice, Second Edititon* (Addison Wesley, 1995 г.).

Для создания и сохранения произвольной кривой в виде последовательности линейных сегментов, кривых второго и третьего порядка предусмотрен класс `GeneralPath`. Для этого нужно передать методу `moveTo()` координату первой точки.

```
GeneralPath path = new GeneralPath();
path.moveTo (10, 20);
```

Затем можно продолжить построение кривой с помощью методов `lineTo()`, `quadTo()` или `curveTo()`. Эти методы продолжают кривую линией, кривой второго порядка или кривой третьего порядка соответственно. Для вызова метода `lineTo()` следует указать конечную точку, а для двух других методов построения кривых нужно сначала указать контрольные точки, а затем конечную.

```
path.lineTo(20, 30);
path.curveTo(control1X, control1Y, control2X, control2Y, endX, endY);
```

Замыкание кривой задается с помощью метода `closePath()`, который рисует линию в обратном направлении вплоть до начальной точки пути.

Чтобы создать многоугольник, нужно вызвать метод `moveTo()` для построения первой вершины, а затем несколько раз вызвать метод `lineTo()` для построения остальных вершин многоугольника. Наконец, нужно вызвать метод `closePath()` для того, чтобы замкнуть многоугольник. Подробности видны из листинга 7.1.

Маршрут не обязательно должен быть связным, т.е. метод `moveTo()` можно вызвать, когда угодно, для начала нового сегмента.

К маршруту можно добавить произвольный объекта типа `Shape`, используя для этой цели метод `append`. При этом контур фигуры добавляется в конце маршрута. Второй параметр метода имеет значение `true`, если новую фигуру необходимо связать с последней точкой траектории, и `false` в противном случае. Например, показанный ниже вызов метода `append` добавляет контур прямоугольника, не соединяя его с концом маршрута:

```
Rectangle2D r = ...;
path.append(r, false);
```

Другой вызов метода `append()` добавляет прямую линию, которая соединяет конечную точку траектории и начальную точку прямоугольника `r`, а затем добавляет контур прямоугольника к полученной конструкции:

```
path.append(r, true);
```

На рис. 7.7 и 7.8 показаны результаты работы программы, код которой приведен в листинге 7.1. В окне этой программы содержится раскрывающийся список для выбора нескольких типов фигур:

- прямые линии;
- прямоугольники, в том числе со скругленными краями, и эллипсы;
- дуги (помимо самой дуги отображаются контуры описанного прямоугольника, маркер начального угла и угол дуги);
- многоугольники (на основе класса `GeneralPath`);
- кривые второго и третьего порядка.

Для изменения расположения контрольных точек следует использовать мышь. Обратите внимание, что при их перемещении фигура автоматически перерисовывается.

Программа имеет довольно сложную структуру, поскольку управляет множеством фигур и поддерживает автоматическую перерисовку фигур при перетаскивании контрольных точек.

Абстрактный суперкласс `ShapeMaker` инкапсулирует общие черты всех классов создания фигур. Каждая фигура имеет контрольные точки, которые пользователь может передвигать, а метод `getPointCount()` возвращает их количество. Для вычисления фактической формы фигуры на основе действительного расположения контрольных точек используется следующий абстрактный метод:

```
Shape make Shape(Point2D[] points);
```

Метод `toString()` возвращает имя класса, которое объект `ShapeMaker` может поместить в объект `JComboBox`.

Для организации перетаскивания контрольных точек в классе `ShapePanel` нужно предусмотреть управление событиями мыши. Если кнопка мыши нажата над прямоугольным маркером контрольной точки, то последующее движение мыши перемещает этот маркер.

Большинство классов, поддерживающих фигуры, имеют очень простую структуру. Их методы `makeShape()` создают и возвращают требуемую фигуру. Однако в классе `ArcMaker` требуется вычислить искаженные начальный и конечный углы. Как показано на рис. 7.9, для демонстрации правильности выполненных вычислений возвращаемая фигура создается на основе класса `GeneralPath`, который описывает саму дугу, описанный вокруг дуги прямоугольник и линии от центра дуги к контрольным точкам, определяющим начальный и конечный угол.

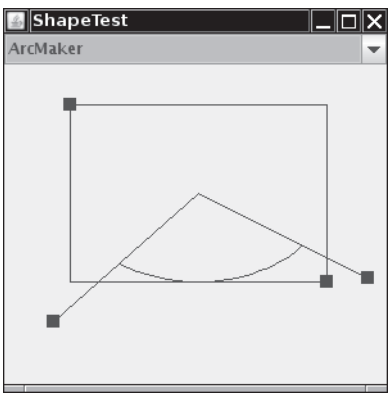


Рис. 7.9. Окно программы `ShapeTest`

Листинг 7.1. Содержимое файла `ShapeTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.awt.geom.*;
4.  import java.util.*;
5.  import javax.swing.*;
6.
7.  /**
8.   * Эта программа демонстрирует примеры
9.   * рисования различных двухмерных фигур.
```

```
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.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.            frame.setVisible(true);
24.         }
25.      });
26.    }
27. }
28.
29. /**
30.  * Фрейм с раскрывающимся списком для выбора
31.  * фигуры и компонентом для ее рисования.
32.  */
33. class ShapeTestFrame extends JFrame
34. {
35.    public ShapeTestFrame()
36.    {
37.       setTitle("ShapeTest");
38.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
39.
40.       final ShapeComponent comp = new ShapeComponent();
41.       add(comp, BorderLayout.CENTER);
42.       final JComboBox comboBox = new JComboBox();
43.       comboBox.addItem(new LineMaker());
44.       comboBox.addItem(new RectangleMaker());
45.       comboBox.addItem(new RoundRectangleMaker());
46.       comboBox.addItem(new EllipseMaker());
47.       comboBox.addItem(new ArcMaker());
48.       comboBox.addItem(new PolygonMaker());
49.       comboBox.addItem(new QuadCurveMaker());
50.       comboBox.addItem(new CubicCurveMaker());
51.       comboBox.addActionListener(new ActionListener()
52.          {
53.             public void actionPerformed(ActionEvent event)
54.             {
55.                ShapeMaker shapeMaker =
56.                   (ShapeMaker) comboBox.getSelectedItem();
57.                comp.setShapeMaker(shapeMaker);
58.             }
59.          });
60.       add(comboBox, BorderLayout.NORTH);
61.       comp.setShapeMaker((ShapeMaker) comboBox.getItemAt(0));
62.    }
63.
64.    private static final int DEFAULT_WIDTH = 300;
65.    private static final int DEFAULT_HEIGHT = 300;
66. }
```

```
67.  /**
68.   * Компонент, отвечающий за рисование фигуры и позволяющий
69.   * пользователю перемещать определяющие ее точки.
70.   */
71.  class ShapeComponent extends JComponent
72.  {
73.     public ShapeComponent()
74.     {
75.        addMouseListener(new MouseAdapter()
76.           {
77.              public void mousePressed(MouseEvent event)
78.              {
79.                 Point p = event.getPoint();
80.                 for (int i = 0; i < points.length; i++)
81.                 {
82.                    double x = points[i].getX() - SIZE / 2;
83.                    double y = points[i].getY() - SIZE / 2;
84.                    Rectangle2D r = new
85.                       Rectangle2D.Double(x, y, SIZE, SIZE);
86.                    if (r.contains(p))
87.                    {
88.                       current = i;
89.                       return;
90.                    }
91.                 }
92.              }
93.
94.              public void mouseReleased(MouseEvent event)
95.              {
96.                 current = -1;
97.              }
98.           });
99.
100.       addMouseMotionListener(new MouseMotionAdapter()
101.          {
102.             public void mouseDragged(MouseEvent event)
103.             {
104.                if (current == -1) return;
105.                points[current] = event.getPoint();
106.                repaint();
107.             }
108.          });
109.       current = -1;
110.    }
111.
112.    /**
113.     * Установка объекта, поддерживающего фигуру, и инициализация
114.     * его набором точек с произвольными координатами.
115.     * @param aShapeMaker Объект, определяющий фигуру на основе
116.     * набора точек
117.     */
118.    public void setShapeMaker(ShapeMaker aShapeMaker)
119.    {
120.       shapeMaker = aShapeMaker;
121.       int n = shapeMaker.getPointCount();
122.       points = new Point2D[n];
```

```
123.      for (int i = 0; i < n; i++)
124.      {
125.         double x = generator.nextDouble() * getWidth();
126.         double y = generator.nextDouble() * getHeight();
127.         points[i] = new Point2D.Double(x, y);
128.      }
129.      repaint();
130.   }
131.
132.   public void paintComponent(Graphics g)
133.   {
134.      if (points == null) return;
135.      Graphics2D g2 = (Graphics2D) g;
136.      for (int i = 0; i < points.length; i++)
137.      {
138.         double x = points[i].getX() - SIZE / 2;
139.         double y = points[i].getY() - SIZE / 2;
140.         g2.fill(new Rectangle2D.Double(x, y, SIZE, SIZE));
141.      }
142.
143.      g2.draw(shapeMaker.makeShape(points));
144.   }
145.
146.   private Point2D[] points;
147.   private static Random generator = new Random();
148.   private static int SIZE = 10;
149.   private int current;
150.   private ShapeMaker shapeMaker;
151. }
152.
153. /**
154.  * Суперкласс классов, определяющих фигуры.
155.  * Метод makeShape() подклассов возвращает фигуру.
156.  */
157. abstract class ShapeMaker
158. {
159.   /**
160.    * Конструктор класса фигуры.
161.    * @param aPointCount Количество точек, необходимых
162.    * для определения данной фигуры
163.    */
164.   public ShapeMaker(int aPointCount)
165.   {
166.      pointCount = aPointCount;
167.   }
168.
169.   /**
170.    * Данный метод возвращает количество точек,
171.    * необходимых для определения фигуры.
172.    * @return Количество точек
173.    */
174.   public int getPointCount()
175.   {
176.      return pointCount;
177.   }
178.
```

```
179.    /**
180.     * Создает фигуру на основе заданного набора точек.
181.     * @param p Точки, определяющие фигуру
182.     * @return Фигура, определенная посредством точек
183.     */
184.    public abstract Shape makeShape(Point2D[] p);
185.
186.    public String toString()
187.    {
188.       return getClass().getName();
189.    }
190.
191.    private int pointCount;
192. }
193.
194. /**
195.  * Создание линии, соединяющей две точки.
196.  */
197. class LineMaker extends ShapeMaker
198. {
199.    public LineMaker()
200.    {
201.      super(2);
202.    }
203.
204.    public Shape makeShape(Point2D[] p)
205.    {
206.       return new Line2D.Double(p[0], p[1]);
207.    }
208. }
209.
210. /**
211.  * Создание прямоугольника, определяемого угловыми точками.
212.  */
213. class RectangleMaker extends ShapeMaker
214. {
215.    public RectangleMaker()
216.    {
217.      super(2);
218.    }
219.
220.    public Shape makeShape(Point2D[] p)
221.    {
222.       Rectangle2D s = new Rectangle2D.Double();
223.       s.setFrameFromDiagonal(p[0], p[1]);
224.       return s;
225.    }
226. }
227.
228. /**
229.  * Создание прямоугольника со скругленными
230.  * углами на основе двух угловых точек.
231.  */
232. class RoundRectangleMaker extends ShapeMaker
233. {
234.    public RoundRectangleMaker()
235.    {
```

```
236.      super(2);
237.   }
238.
239.   public Shape makeShape(Point2D[] p)
240.   {
241.      RoundRectangle2D s = new
242.         RoundRectangle2D.Double(0, 0, 0, 0, 20, 20);
243.      s.setFrameFromDiagonal(p[0], p[1]);
244.      return s;
245.   }
246. }
247.
248. /**
249.  * Создание эллипса, ограничивающий прямоугольник
250.  * которого задается посредством двух угловых точек.
251.  */
252. class EllipseMaker extends ShapeMaker
253. {
254.   public EllipseMaker()
255.   {
256.    super(2);
257.   }
258.
259.   public Shape makeShape(Point2D[] p)
260.   {
261.      Ellipse2D s = new Ellipse2D.Double();
262.      s.setFrameFromDiagonal(p[0], p[1]);
263.      return s;
264.   }
265. }
266.
267. /**
268.  * Создание дуги, ограничивающий прямоугольник которой строится
269.  * на основании двух угловых точек. Начальный угол и угол дуги
270.  * задаются с помощью прямых, проведенных через центр
271.  * ограничивающего прямоугольника и соответствующую
272.  * контрольную точку. Для демонстрации корректности
273.  * процесса построения дуги отображаются ограничивающий
274.  * прямоугольник и линии, определяющие углы.
275.  */
276. class ArcMaker extends ShapeMaker
277. {
278.   public ArcMaker()
279.   {
280.    super(4);
281.   }
282.
283.   public Shape makeShape(Point2D[] p)
284.   {
285.      double centerX = (p[0].getX() + p[1].getX()) / 2;
286.      double centerY = (p[0].getY() + p[1].getY()) / 2;
287.      double width = Math.abs(p[1].getX() - p[0].getX());
288.      double height = Math.abs(p[1].getY() - p[0].getY());
289.
290.     double skewedStartAngle=
291.        Math.toDegrees(Math.atan2(-(p[2].getY() - centerY)
292.        * width, (p[2].getX() - centerX) * height));
```

```
293.      double skewedEndAngle =
294.        Math.toDegrees(Math.atan2(-(p[3].getY() - centerY)
295.        * width, (p[3].getX() - centerX) * height));
296.      double skewedAngleDifference =
297.        skewedEndAngle - skewedStartAngle;
298.      if (skewedStartAngle < 0) skewedStartAngle += 360;
299.      if (skewedAngleDifference < 0)
300.        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.  * Создание многоугольника, определяемого 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.  * Создание кривой второго порядка, определенной
341.  * двумя конечными точками и контрольной 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.  * Создание кривой третьего порядка, определенной
359.  * двумя конечными и двумя контрольными 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(), p[2].getY(),
372.        p[3].getX(), p[3].getY());
373.   }
374. }
```

API java.awt.geom.RoundRectangle2D.Double 1.2

- `RoundRectangle2D.Double(double x, double y, double width, double height, double arcWidth, double arcHeight)`

Создает прямоугольник со скругленными углами на основе параметров ограничивающего прямоугольника и дуги скругления. То, что собой представляют параметры `arcWidth` и `arcHeight`, можно увидеть на рис. 7.4.

API java.awt.geom.Arc2D.Double 1.2

- `Arc2D.Double(double x, double y, double w, double h, double startAngle, double arcAngle, int type)`

Создает дугу по заданному ограничивающему прямоугольнику, начальному и конечному углу и типу замыкания. Объяснение того, что собой представляют параметры `startAngle` и `arcAngle`, уже приводилось ранее. Что касается параметра типа, то для него может указываться одно из следующих значений: `Arc2D.OPEN`, `Arc2D.PIE` или `Arc2D.CHORD`.

API java.awt.geom.QuadCurve2D.Double 1.2

- `QuadCurve2D.Double(double x1, double y1, double ctrlx, double ctrly, double x2, double y2)`

Создает кривую второго порядка по начальной, контрольной и конечной точкам.

API 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.

API java.awt.geom.GeneralPath 1.2

- `GeneralPath()`

  Создает пустую кривую.

API java.awt.geom.Path2D.Float 6

- `void moveTo(float x, float y)`

  Создает *текущую точку* с координатами `x` и `y`, т.е. начальную точку следующего сегмента.

- `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)`

  Рисует прямую линию, линию второго или третьего порядка от текущей точки к конечной с координатами `x` и `y`, после чего конечная точка становится текущей.

API java.awt.geom.Path2D 6

- • `void append(Shape s, boolean connect)`

  Добавляет контур данной фигуры в состав траектории. Если значение `connect` равно `true`, то текущая точка траектории соединяется прямой линией с начальной точкой добавленной фигуры.

- `void closePath()`

  Замыкает путь, рисуя прямую линию от текущей точки к первой точке траектории.

Области

В предыдущем разделе описаны способы создания 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 принтере, состоят из прямых и кривых третьего порядка.

Однако иногда легче описать фигуру, объединяя *области*, т.е. прямоугольники, многоугольники или эллипсы. В Java 2D API поддерживаются четыре *метода обработки областей*, которые объединяют две области в одну.

- Метод `add()` создает область, которая содержит все точки первой и второй фигуры.
- Метод `subtract()` создает область, которая содержит все точки первой фигуры, не входящие во вторую фигуру.

- Метод `intersect()` создает область, которая содержит только точки, принадлежащие одновременно первой и второй фигуре.
- Метод `exclusiveOr()` создает область, которая содержит все точки, принадлежащие либо первой, либо второй фигуре, но не обеим одновременно.

Результаты применения этих методов показаны на рис. 7.10.

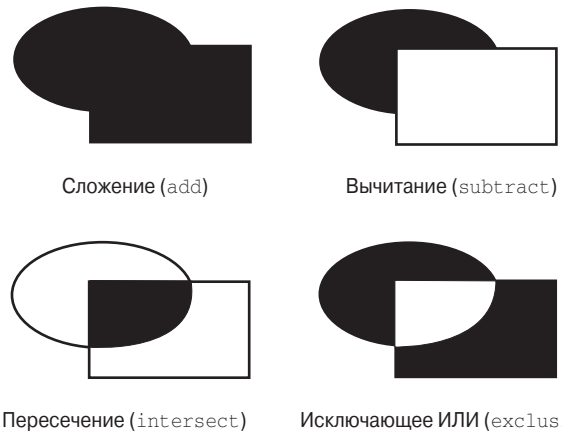


Рис. 7.10. Результаты применения различных методов обработки областей

Для создания сложной фигуры нужно создать начальную область:

```
Area a = new Area();
```

Затем следует объединить эту область с фигурами нужной формы:

```
a.add(new Rectangle2D.Double(...));
a.subtract(path);
...
```

Класс `Area` реализует интерфейс `Shape`. В процессе выполнения программы можно либо очертить границу области с помощью метода `draw()`, либо заполнить внутреннюю часть фигуры с помощью метода `fill()` класса `Graphics2D`.

API java.awt.geom.Area

- `void add(Area other)`
- `void subtract(Area other)`
- `void intersect(Area other)`
- `void exclusiveOr(Area other)`

  Выполняет операцию объединения текущей области с областью `other`. В результате выполнения операции модифицируется текущая область.

Штрихи

Метод `draw()` класса `Graphics2D` рисует границу фигуры с использованием выбранного *типа линии*, или *штриха* (stroke). По умолчанию используется сплошная линия толщиной в один пиксель. Для изменения типа штриха предусмотрен метод `setStroke()`, которому в качестве параметра передается экземпляр класса, реализую-

щего интерфейс `Stroke`. В Java 2D API определен только один такой класс, а именно — `BasicStroke`. В этом разделе рассматриваются возможности и способы использования данного класса.

Для штриха можно задавать линии произвольной толщины. Например, для рисования линии толщиной 10 пикселей можно применить следующий код:

```
g2.setStroke(new BasicStroke(10.0F));
g2.draw(new Line2D.Double(...));
```

Если толщина линии превышает один пиксель, то для формирования *концов* линии можно использовать один из перечисленных ниже стилей (рис. 7.11).

- *Стык* (butt cap). Линия обрывается в конечной точке.
- *Овальное перекрытие* (round cap). В конце штриха добавляется полукруг.
- *Прямоугольное перекрытие* (square cap). В конце штриха добавляется прямоугольник.

Для соединения двух штрихов толщиной более пикселя также можно указать один из следующих стилей (рис. 7.12).

- *Косое соединение* (bevel join). Соединяет штрихи по прямой линии, перпендикулярной биссектрисе угла между ними.
- *Скругленное соединение* (round join). Соединяет штрихи с помощью овального перекрытия.
- *Фасетное соединение* (miter join). Продолжает штрихи с образованием клина.

Фасетное соединение не очень подходит для линий, которые пересекаются под небольшими углами. Поэтому если две линии пересекаются под углами, меньшими *предельного фасетного угла* (miter limit), то вместо фасетного используется косое соединение. Это предотвращает образование чрезмерно длинных клиньев. По умолчанию предельный фасетный угол равен 10°. Стили окончания и пересечения штрихов можно указать в конструкторе `BasicStroke`:

```
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND));
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_BUTT, BasicStroke.JOIN_MITTER,
   15.0F /* предельный фасетный угол */));
```



Рис. 7.11. Стили окончания линий



Рис. 7.12. Стили пересечения штрихов

Наконец, для пунктирных штрихов можно выбрать *шаблон пунктира* (dash pattern). В листинге 7.2 используется шаблон пунктира, который соответствует сигналу “SOS” в азбуке Морзе. Шаблон пунктира задается в виде массива `float[]`, который содержит длины рисуемых (on) и не рисуемых (off) фрагментов штриха (рис. 7.13).



Рис. 7.13. Пример шаблона пунктира

При создании `BasicStroke` помимо шаблона пунктира указывается также *фаза пунктира* (dash phase), т.е. место начала шаблона в данной линии. Обычно фаза имеет значение 0.

```
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 /* предельный фасетный угол */,
   dashPattern,
   0 /* фаза пунктира */));
```



Стили окончания линий применяются для *всех* элементов пунктира.

Программа, код которой представлен в листинге 7.2, позволяет указать стили окончания штриха, стиль соединения штрихов и тип пунктира (рис. 7.14). Кроме того, можно перемещать концы штрихов для проверки учета предельного фасетного угла. Для этого выберите фасетное пересечение, а затем перетащите мышью сегмент так, чтобы две линии образовывали очень острый угол. После достижения предельного фасетного угла фасетное соединение автоматически превратится в косое.

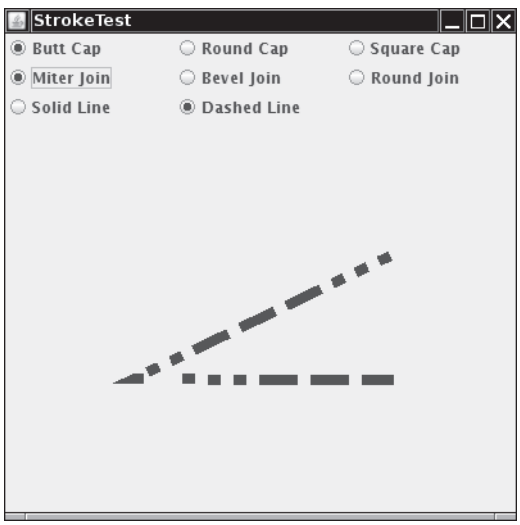


Рис. 7.14. Окно программы `StrokeTest`

Данная программа похожа на программу, код которой приведен в листинге 7.1. Обработчик событий мыши реагирует на щелчок в конечной точке штриха, и при движении курсора мыши будет перемещать вслед за ним конец штриха. Набор переключателей показывает настройки окончания линии, стиля соединения и типа пунктира. Для создания всей линии `GeneralPath` используется метод `paintComponent()` класса `StrokePanel`. Вся линия состоит из двух отрезков, соединяющих три точки, которые пользователь может перемещать с помощью мыши. На основании установок, сделанных пользователем, создается объект `BasicStroke`, после чего линия отображается на экране.

Листинг 7.2. Содержимое файла `StrokeTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.awt.geom.*;
4.  import java.util.*;
5.  import javax.swing.*;
6.
7.  /**
8.   * Эта программа демонстрирует использование штрихов различных типов.
9.   * @version 1.03 2007-08-16
10.  * @author Cay Horstmann
11.  */
12. public class StrokeTest
13. {
14.    public static void main(String[] args)
15.    {
16.      EventQueue.invokeLater(new Runnable()
17.       {
18.         public void run()
19.         {
20.           JFrame frame = new StrokeTestFrame();
21.           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22.           frame.setVisible(true);
23.         }
24.     });
25.   }
26. }
27.
28. /**
29.  * Фрейм с набором переключателей
30.  * для выбора стиля окончания штриха,
31.  * стиля пересечения штрихов и
32.  * типа линии.
33.  */
34. class StrokeTestFrame extends JFrame
35. {
36.    public StrokeTestFrame()
37.    {
38.       setTitle("StrokeTest");
39.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
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,
55.         group2);
56.       makeJoinButton("Bevel Join", BasicStroke.JOIN_BEVEL,
57.         group2);
58.       makeJoinButton("Round Join", BasicStroke.JOIN_ROUND,
59.         group2);
60.
61.       ButtonGroup group3 = new ButtonGroup();
62.       makeDashButton("Solid Line", false, group3);
63.       makeDashButton("Dashed Line", true, group3);
64.    }
65.
66.    /**
67.     * Создание переключателей для выбора стиля окончания штриха.
68.     * @param label Надпись рядом с переключателем
69.     * @param style Стиль окончания штриха
70.     * @param group Группа переключателей
71.     */
72.    private void makeCapButton(String label, final int style,
73.       ButtonGroup group)
74.    {
75.       // Выбор первого переключателя в группе.
76.       boolean selected = group.getButtonCount() == 0;
77.       JRadioButton button = new JRadioButton(label, selected);
78.       buttonPanel.add(button);
79.       group.add(button);
80.       button.addActionListener(new ActionListener()
81.          {
82.             public void actionPerformed(ActionEvent event)
83.             {
84.                canvas.setCap(style);
85.             }
86.          });
87.    }
88.
89.    /**
90.     * Создание переключателей для выбора стиля соединения линий.
91.     * @param label Надпись рядом с переключателем
92.     * @param style Стиль соединения линий
93.     * @param group Группа переключателей
94.     */
95.    private void makeJoinButton(String label, final int style,
96.       ButtonGroup group)
97.    {
98.       // Выбор первого переключателя в группе.
99.       boolean selected = group.getButtonCount() == 0;
100.      JRadioButton button = new JRadioButton(label, selected);
101.      buttonPanel.add(button);
```

```
102.      group.add(button);
103.      button.addActionListener(new ActionListener()
104.         {
105.            public void actionPerformed(ActionEvent event)
106.            {
107.               canvas.setJoin(style);
108.            }
109.         });
110.   }
111.
112.   /**
113.    * Создание переключателей для выбора сплошной
114.    * или пунктирной линии.
115.    * @param label Надпись рядом с переключателем
116.    * @param style false для сплошной линии, true для пунктирной
117.    * @param group Группа переключателей
118.    */
119.   private void makeDashButton(String label, final boolean style,
120.      ButtonGroup group)
121.   {
122.      // Выбор первого переключателя в группе.
123.      boolean selected = group.getButtonCount() == 0;
124.      JRadioButton button = new JRadioButton(label, selected);
125.      buttonPanel.add(button);
126.      group.add(button);
127.      button.addActionListener(new ActionListener()
128.         {
129.            public void actionPerformed(ActionEvent event)
130.            {
131.               canvas.setDash(style);
132.            }
133.         });
134.   }
135.
136.   private StrokePanel canvas;
137.   private JPanel buttonPanel;
138.
139.   private static final int DEFAULT_WIDTH = 400;
140.   private static final int DEFAULT_HEIGHT = 400;
141. }
142.
143. /**
144.  * Компонент, в котором рисуются две соединенные
145.  * линии с использованием разных объекты штрихов и
146.  * который позволяет пользователю перетаскивать три
147.  * определяющих расположение линий точки.
148.  */
149. class StrokeComponent extends JComponent
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
163.                 Rectangle2D.Double(x, y, SIZE, SIZE);
164.               if (r.contains(p))
165.               {
166.                 current = i;
167.                 return;
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.         });
186.       points = new Point2D[3];
187.       points[0] = new Point2D.Double(200, 100);
188.       points[1] = new Point2D.Double(100, 200);
189.       points[2] = new Point2D.Double(200, 200);
190.       current = -1;
191.       width = 8.0F;
192.     }
193.
194.     public void paintComponent(Graphics g)
195.     {
196.       Graphics2D g2 = (Graphics2D) g;
197.       GeneralPath path = new GeneralPath();
198.       path.moveTo((float) points[0].getX(),
199.         (float) points[0].getY());
200.       for (int i = 1; i < points.length; i++)
201.         path.lineTo((float) points[i].getX(),
202.           (float) points[i].getY());
203.       BasicStroke stroke;
204.       if (dash)
205.       {
206.         float miterLimit = 10.0F;
207.         float[] dashPattern = { 10F, 10F, 10F, 10F, 10F, 10F,
208.           30F, 10F, 30F, 10F, 30F, 10F, 10F, 10F, 10F, 10F,
209.           10F, 30F };
210.         float dashPhase = 0;
211.         stroke = new BasicStroke(width, cap, join,
212.           miterLimit, dashPattern, dashPhase);
213.       }
214.       else stroke = new BasicStroke(width, cap, join);
```

```
215.      g2.setStroke(stroke);
216.      g2.draw(path);
217.    }
218.    /**
219.     * Установка стиля соединения линий.
220.     * @param j Стиль соединения линий
221.     */
222.    public void setJoin(int j)
223.    {
224.      join = j;
225.      repaint();
226.    }
227.    /**
228.     * Установка стиля окончания линии.
229.     * @param c Стиль окончания линии
230.     */
231.    public void setCap(int c)
232.    {
233.      cap = c;
234.      repaint();
235.    }
236.    /**
237.     * Переключение между сплошной и пунктирной линией.
238.     * @param d false для сплошной, true для пунктирной линии
239.     */
240.    public void setDash(boolean d)
241.    {
242.      dash = d;
243.      repaint();
244.    }
245.    private Point2D[] points;
246.    private static int SIZE = 10;
247.    private int current;
248.    private float width;
249.    private int cap;
250.    private int join;
251.    private boolean dash;
252. }
```

API java.awt.Graphics2D 1.2

- `void setStroke(Stroke s)`

  Задает параметры штриха графического контекста для объекта, который реализует интерфейс `Stroke`.

API 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)`

  Создает объект штриха с заданными характеристиками.

Параметры:	`width`	Толщина пера
	`cap`	Стиль окончания линии: `CAP_BUTT`, `CAP_ROUND` или `CAP_SQUARE`
	`join`	Стиль соединения линий: `JOIN_BEVEL`, `JOIN_MITER` или `JOIN_ROUND`
	`miterlimit`	Предельный фасетный угол (в градусах); если угол между пересекающимися линиями меньше его, то фасетное соединение превращается в косое
	`dash`	Шаблон пунктира; массив длин для отображаемых и скрытых фрагментов
	`dashPhase`	Фаза пунктира – длина сегмента, якобы предшествующего первой точке линии; предполагается, что шаблон применяется с начала этого сегмента

Расцветки

Внутренняя часть фигуры заполняется в соответствии с *расцветкой* (paint). Для указания стиля расцветки используется объект, класс которого реализует интерфейс `Paint`. Этот объект передается методу `setPaint()`. В Java 2D API предусмотрено три таких класса.

- Класс `Color` предназначен для заполнения фигуры сплошным цветом. Нужно лишь вызвать метод `setPaint()` и указать объект `Color`:

```
g2.setPaint(Color.red);
```

- Класс `GradientPaint` предназначен для заполнения фигуры градиентной заливкой, т.е. постепенно изменяющимся цветом (рис. 7.15).
- Класс `TexturePaint` предназначен для заполнения фигуры мозаичной текстурой, т.е. несколькими копиями рисунков (рис. 7.16).

Для создания объекта `GradientPaint` нужно указать две точки и цвета для них:

```
g2.setPaint(new GradientPaint(p1, Color. RED, p2, Color.YELLOW));
```

Изменение цвета осуществляется вдоль прямой, соединяющей эти точки. Цвета остаются постоянными вдоль прямых, перпендикулярных к линии соединения. Для точек, которые находятся за концами соединительной линии, задаются цвета соответствующих конечных точек.

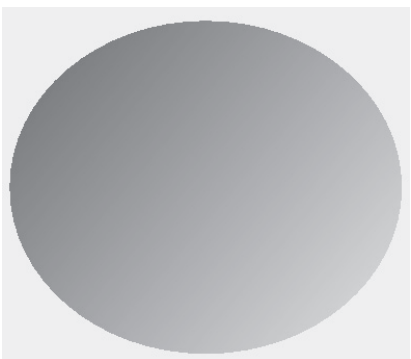


Рис. 7.15. Градиентное заполнение

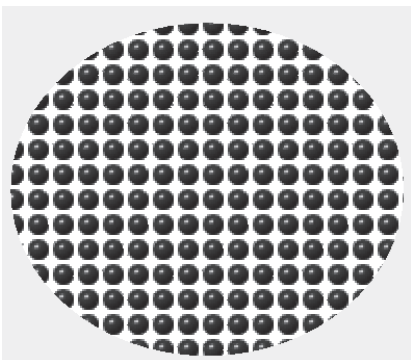


Рис. 7.16. Мозаичная текстура

Если вызвать конструктор объекта `GradientPaint` с параметром `cyclic`, имеющим значение `true`, то изменение цвета будет *циклически* происходить и за пределами конечных точек:

```
g2.setPaint(new GradientPaint(p1, Color.RED, p2, Color.YELLOW, true));
```

Для создания объекта-текстуры `TexturePaint` нужно указать копируемый объект-рисунок `BufferImage` и *прямоугольник привязки* (anchor rectangle):

```
g2.setPaint(new TexturePaint(bufferedImage, anchorRectangle));
```

Класс `BufferedImage` будет более подробно рассматриваться чуть позже в этой главе, при детальном обсуждении способов работы с рисунками. Пока что достаточно знать лишь то, что наиболее простым способом для получения буферизованного изображения (`BufferedImage`) является считывание файла изображения:

```
bufferedImage = ImageIO.read(new File("blue-ball.gif"));
```

Прямоугольник привязки повторяется бесконечное количество раз в направлениях, параллельных осям x и y с образованием мозаики. Заданный объект-рисунок масштабируется таким образом, чтобы соответствовать размерам прямоугольника привязки, а затем повторяется в каждом элементе мозаики.

API java.awt.Graphics2D 1.2

- `void setPaint(Paint s)`

  Задает тип расцветки для графического контекста с помощью объекта, который реализует интерфейс `Paint`.

API 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)`

  Создает градиентную расцветку. При ее использовании фигура заполняется таким образом, что начальная точка с координатами `x1` и `y1` имеет цвет `color1`, конечная точка с координатами `x2` и `y2` имеет цвет `color2`, а между ними цвет изменяется линейно. Цвета остаются постоянными вдоль прямой, перпендикулярной линии, соединяющей начальную и конечные точки.

API java.awt.TexturePaint 1.2

- `TexturePaint(BufferedImage texture, Rectangle2D anchor)`

  Создает объект текстурной расцветки. Прямоугольник привязки задает мозаичную фрагментацию подлежащей раскрашиванию области; он повторяется бесконечное число раз как в направлении x, так и в направлении y, и масштабирует заданное изображение текстуры так, чтобы оно заполняло каждый элемент мозаики.

Преобразование координат

Предположим, нужно нарисовать автомобиль по известным параметрам (высота, расстояние между осями, общая длина машины и т.д.), которые выражены в метрах. Конечно, можно вычислять все координаты рисуемых частей автомобиля в пикселях, подсчитав количество пикселей на метр. Однако все это можно сделать намного проще с помощью преобразования координат:

```
g2.scale(pixelsPerMeter, pixelsPerMeter);
g2.draw(new Line2D.Double(координаты_в_метрах));
   // Преобразование размеров, выраженных в метрах,
   // в пиксели и рисование линии
```

Метод `scale()` класса `Graphics2D` выполняет *преобразование координат* графического контекста. При этом *пользовательские координаты* (единицы измерения, указанные пользователем) преобразуются в *аппаратные координаты* (пиксели). На рис. 7.17 показан пример такого преобразования.

Преобразование координат очень удобно применять на практике, так как оно позволяет работать с общепринятыми единицами измерения. Графический контекст автоматически позаботится о переводе их в пиксели.

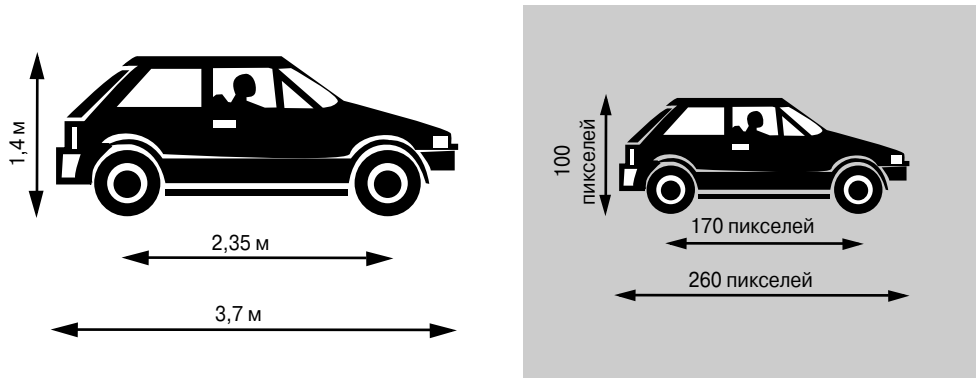


Рис. 7.17. Пользовательские и аппаратные координаты

Платформа Java предусматривает четыре основных типа преобразований координат.

- Масштабирование: растяжение или сжатие всех расстояний от фиксированной точки.
- Поворот: вращение всех точек фигуры вокруг фиксированной точки.
- Перемещение: смещение 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 применения перечисленных выше типов преобразования к квадрату.

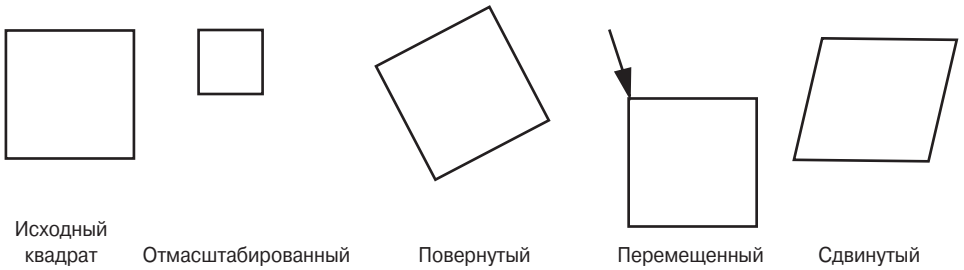


Рис. 7.18. Основные типы преобразования координат

Методы `scale()`, `rotate()`, `translate()` и `shear()` класса `Graphics2D` реализуют перечисленные различные типы преобразования.

Эти методы можно *комбинировать*, например, повернуть фигуру *и* удвоить ее размер. Необходимо лишь выполнить операции поворота и масштабирования:

```
g2.rotate(angle);
g2.scale(2, 2);
g2.draw(...);
```

В данном случае не имеет значения, в какой последовательности будут выполняться операции преобразования. Однако порядок применения большинства операций имеет значение. Например, если нужно повернуть и сдвинуть изображение, необходимо решить, какое преобразование сделать первым, чтобы добиться необходимого результата. Графический контекст выполняет преобразования в порядке, обратном тому, в котором они были указаны. Это значит, что последнее заданное преобразование выполняется первым.

Допустимо любое количество преобразований координат. Рассмотрим следующую последовательность вызовов:

```
g2.translate(x, y);
g2.rotate(a);
g2.translate(-x, –y);
```

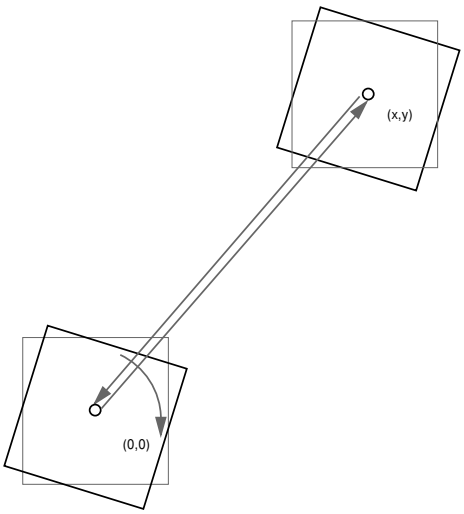


Рис. 7.19. Сочетание различных методов преобразования координат

Последнее преобразование (которое выполняется первым) перемещает всю фигуру так, что точка (`x,y`) попадает в начало координат. Второе преобразование поворачивает фигуру на угол `a` вокруг начала координат. Последнее преобразование снова перемещает всю фигуру так, что начало координат попадает в точку с координатами (`x,y`). В итоге (рис. 7.19) получаем поворот фигуры вокруг точки с координатами (`x,y`). Поскольку поворот фигуры вокруг точки, отличной от начала координат, выполняется довольно часто, для этой операции предусмотрен специальный метод:

```
g2.rotate(a, x, y);
```

Известно, что поворот, масштабирование, перемещение, сдвиг и их сочетания могут быть выражены в виде преобразования матриц:

$$\begin{bmatrix} x_{new} \\ y_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Такие преобразования называются *аффинными* и в Java 2D API описываются с помощью класса `AffineTransform`. Объект аффинного преобразования можно непосредственно создать, если известны компоненты матрицы преобразования:

```
AffineTransform t = new AffineTransform (a, b, c, d, e, f);
```

Для реализации отдельных типов преобразований предусмотрены также методы `getRotateInstance()`, `getScaleInstance()`, `getTranslateInstance()` и `getShearInstance()`, назначение которых можно легко определить по их названию. Например, вызов метода

```
t = AffineTransform.getScaleInstance(2.0F, 0.5F)
```

возвращает преобразование, которое соответствует матрице

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Наконец, методы экземпляра `setToRotate()`, `setToScale()`, `setToTranslation()` и `setToShear()` предназначены для указания нового типа преобразования:

```
t.setToRotation(angle);  // Установка поворота
```

Для замены текущего преобразования аффинным преобразованием координат графического контекста предусмотрен метод `setTransform()`:

```
g2.setTransform(t);      // Замещает текущее преобразование
```

Однако на практике не рекомендуется вызывать операцию `setTransform()`, поскольку она неправильно интерпретирует все существующие усеченные фигуры. Например, графический контекст для печати в альбомной ориентации страницы уже содержит преобразование-поворот на 90°. При вызове `setTransform()` это преобразование-поворот игнорируется. В таком случае вместо метода `setTransform()` рекомендуется вызвать метод `transform()`:

```
g2.transform(t);      // комбинирует текущее преобразование с преобразованием t
```

Данный метод объединяет текущее преобразование с новым объектом аффинного преобразования.

Если нужно применить какое-то преобразование временно, то следует сначала сохранить предыдущее преобразование, затем объединить его с новым и, наконец, восстановить старое преобразование:

```
AffineTransform oldTransform = g2.getTransform();
   // сохранение старого преобразование

g2.transform(t); // выполнение временного преобразования
   // рисование в графическом контексте g2

g2.setTransform(oldTransform); //восстановление старого преобразования
```

API 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)`

  Формирует аффинное преобразование в соответствии с приведенной ниже матрицей.

$$\begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix}$$

- `AffineTransform(double[] m)`
- `AffineTransform(float[] m)`

  Формирует аффинное преобразование в соответствии с представленной ниже матрицей.

$$\begin{bmatrix} m[0] & m[2] & m[4] \\ m[1] & m[3] & m[5] \\ 0 & 0 & 1 \end{bmatrix}$$

- `static AffineTransform getRotateInstance(double a)`

  Создает преобразование поворота против часовой стрелки вокруг начала координат на угол `a` (заданный в радианах). Матрица преобразования для этого случая показана ниже.

$$\begin{bmatrix} \cos(a) & -\sin(a) & 0 \\ \sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

  Если значение a находится в пределах от 0 до $\pi/2$, то в результате преобразования положительное направление оси x перемещается к положительному направлению оси y.

- `static AffineTransform getRotateInstance(double a, double x, double y)`

  Создает преобразование поворота против часовой стрелки на угол `a` (заданный в радианах) вокруг точки с координатами (`x`, `y`).

- `static AffineTransform getScaleInstance(double sx, double sy)`

  Создает преобразование масштабирования с коэффициентами масштабирования `sx` и `sy` по осям x и y соответственно. Матрица преобразования для этого случая показана ниже.

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- `static AffineTransform getShearInstance(double shx, double shy)`

  Создает преобразование сдвига по осям x и y, определяемого коэффициентами `shx` и `shy`. Матрица преобразования для этого случая показана ниже.

$$\begin{bmatrix} 1 & shx & 0 \\ shy & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- `static AffineTransform getTranslateInstance(double tx, double ty)`
  Создает преобразование перемещения по осям x и y на значения `tx` и `ty`.

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}$$

- `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)`
  Устанавливает аффинное преобразование в соответствие с указанными параметрами. Параметры интерпретируются так же, как и в методах `get`*Xxx*`Instance()`.

API **java.awt.Graphics2D 1.2**

- `void setTransform(AffineTransform t)`
  Заменяет существующее преобразование координат данного графического контекста преобразованием `t`.
- `void transform(AffineTransform t)`
  Объединяет существующее преобразование координат данного графического контекста с 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)`
  Объединяет существующее преобразование координат данного графического контекста с указанным преобразованием, определяемым набором параметров. Параметры интерпретируются так же, как и в методах `AffineTransform.get`*Xxx*`Instance()`.

Отсечение

Для выполнения всех графических операций только внутри заданной области предусмотрено использование *фигуры отсечения* (clipping shape) графического контекста:

```
g2.setClip(clipShape); // Лучше использовать метод clip, подробнее см. ниже.
g2.draw(shape);  // Рисует только фрагменты, находящиеся внутри фигуры отсечения.
```

Однако на практике не рекомендуется вызывать операцию `setClip()`, так как она заменяет все существующие фигуры отсечения данного графического контекста. Например, как показано далее, графический контекст для печати уже содержит прямоугольник отсечения, который позволяет избежать появления данных на полях страницы. В этом случае вместо метода `setClip()` нужно вызвать метод `clip()`:

```
g2.clip(clipShape); // Более удачный способ
```

Метод clip() формирует пересечения существующей фигуры отсечения с указанной фигурой.

Если необходимо применить фигуру отсечения только временно, сначала потребуется сохранить старую фигуру, затем добавить новую и, наконец, восстановить старую фигуру отсечения.

```
Shape oldClip = g2.getClip();  // Сохранение старой фигуры отсечения
g2.clip(clipShape);            // Установка временной фигуры
// Операции рисования в графическом контексте g2
g2.setClip(oldClip);           // Восстанавление старой фигуры
```

На рис. 7.20 возможность отсечения показана на примере рисования довольно непростого шаблона линий, который отсекается сложной фигурой, а именно — контуром ряда символов.



Рис. 7.20. Использование фигур символов для отсечения шаблона линий

Для получения контуров символов нужно получить *контекст отображения шрифтов* (font render context) с помощью метода getFontRenderContext() класса Graphics2D:

```
FontRenderContext context = g2.getFontRenderContext();
```

Затем, используя строку, шрифт и контекст отображения шрифтов, следует создать объект TextLayout:

```
TextLayout layout = new TextLayout("Hello", font, context);
```

Данный объект описывает последовательность символов, внешний вид которой будет зависеть от выбранного контекста отображения. Как известно, одни и те же символы могут по-разному выглядеть на экране или на странице, полученной с помощью принтера.

Более важным является то, что метод getOutline() возвращает объект Shape, который описывает фигуру контура символов. Эта фигура начинается в точке (0, 0), что может и не быть подходящим вариантом, в случае чего для операции getOutline() следует использовать аффинное преобразование, позволяющее указать то, в какой именно точке должен появляться контур.

```
AffineTransform transform = AffineTransform.getTranslateInstance(0,100);
Shape outline = layout.getOutline(transform);
```

Далее нужно добавить контур к фигуре отсечения:

```
GeneralPath clipShape = new GeneralPath();
clipShape.append(outline, false);
```

И, наконец, последнее, что необходимо сделать — это установить фигуру отсечения и нарисовать набор линий. После этого линии будут появляться только внутри границ 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)); // Линии отсекаются по контуру
}
```

Весь полностью код можно найти далее в этой главе, в листинге 7.8.

API 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.

API java.awt.Graphics2D 1.2

- `void clip(Shape s)`

  Создает пересечение текущей фигуры отсечения с фигурой `s`.

- `FontRenderContext getFontRenderContext()`

  Возвращает контекст отображения шрифтов, необходимый для создания объекта `TextLayout`.

API java.awt.font.TextLayout 1.2

- `TextLayout(String s, Font f, FontRenderContext context)`

  Создает объект `TextLayout` на основе заданной строки, шрифта и контекста отображения.

- `Float getAdvance()`

  Возвращает ширину макета, определяемого объектом `TextLayout`.

- `float getAscent()`
- `float getDescent()`

  Возвращает высоту и глубину посадки символов относительно базовой линии.

- `Float getLeading()`

  Возвращает расстояние между соседними строками для шрифта, используемого в данном макете.

Прозрачность и композиция

В стандартной модели RGB каждый цвет описывается на основе трех основных цветов: красного, зеленого и синего. Иногда бывает удобно использовать *прозрачные* (transparent) или частично прозрачные изображения. При наложении рисунка на уже существующее изображение прозрачные пиксели не закрывают пикселей, находящихся под ними, а частично прозрачные пиксели смешиваются с нижележащими пикселями.

На рис. 7.21 показан эффект наложения частично прозрачного прямоугольника на уже имеющееся изображение. При этом элементы рисунка, находящиеся под прямоугольником, хорошо видны.



Рис. 7.21. Наложение частично прозрачного прямоугольника на рисунок

В Java 2D API прозрачность описывается с помощью *альфа-канала* (alpha channel). Каждый пиксель, кроме красной, зеленой и синей составляющей, имеет альфа-значение в интервале от 0 (полностью прозрачен) до 1 (не прозрачен). Например, прямоугольник на рис. 7.21 заполнен бледно-желтым цветом с прозрачностью 50%:

```
new Color(0.7F, 0.7F, 0.0F, 0.5F);
```

Теперь посмотрим, что произойдет при наложении двух фигур. При этом нужно смешать или *скомпоновать* цвета и альфа-значения соответствующих пикселей. Исследователи Т. Портер (T. Porter) и Т. Дафф (T. Duff), работающие в области компьютерной графики, сформулировали 12 возможных правил композиции, которые реализованы в Java 2D API. Следует отметить, что только два из них имеют практическое применение. Если эти правила покажутся слишком сложными, то вместо них рекомендуется использовать простое правило `SRC_OVER`, которое применяется по умолчанию для объектов `Graphics2D` и дает интуитивно понятные результаты.

Рассмотрим кратко основы этих правил. Допустим, что *пиксель источника* (source pixel) имеет альфа-значение a_s, а *целевой пиксель* (destination pixel) — альфа-значение a_D. На рис. 7.22 показана диаграмма, поясняющая правила композиции для этих значений.

Портер и Дафф считают, что альфа-значение выражает вероятность использования цвета пикселя при объединении изображений. Для пикселя источника с вероятностью a_s исходный цвет будет использоваться, а с вероятностью $1-a_s$ — не будет. Эти же соображения верны и для целевого пикселя. Предположим, что при объединении цветов эти вероятности являются независимыми. На рис. 7.22 показаны четыре возможных случая. Если нужно преимущественно использовать цвет пикселя источника, а не целевого пикселя (эта ситуация обозначается символом S), то вероятность этого события равна $a_s(1-a_d)$. Аналогично вычисляется вероятность $a_d(1-a_s)$ преимущественного использования цвета целевого пикселя, а не пикселя источника (эта ситуация обозначается символом D). Что же делать, если источник и целевое изображение "считают" свой цвет преимущественным? Именно в этой ситуации и используются правила Портера–Дафа. Если предпочтение отдается цвету источника, то в таком случае правый нижний угол диаграммы помечается буквой S, а само правило называется `SRC_OVER`. В нем комбинируются цвет источника с весом a_s и целевой цвет с весом $a_d(1-a_s)$.

Визуальным эффектом применения этого правила является смешение пикселя источника и целевого пикселя с приоритетом для пикселя источника. В частности, если $a_s = 1$, то цвет целевого пикселя вообще не принимается в расчет. А если $a_s = 0$, то пиксель источника полностью прозрачен и цвет целевого пикселя не изменяется.

В зависимости от того, как расставлены буквы на диаграмме, можно сформировать другие правила. Они перечислены табл. 7.1 и условно показаны на рис. 7.22. Изображения на рисунке представляют результаты выполнения этих правил для прямоугольной области изображения-источника с альфа-значением 0,75 и эллиптической области целевого изображения с альфа-значением 1,0.

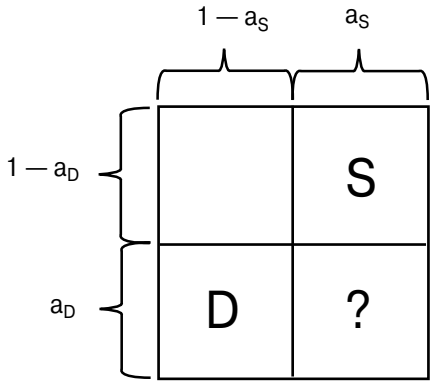


Рис. 7.22. Разработка правила композиции

Как можно заметить, большинство этих правил вряд ли применяются на практике. Так, например, правило `DST_IN` представляет собой крайний случай, в котором вовсе не принимается в расчет цвет пикселя-источника, но используется альфа-канал пикселя-источника для изменения целевого пикселя. В отличие от него, правило `SRC` очень полезно, потому что заставляет использовать цвет источника, выключая смешение с цветом целевого изображения.

Более подробно о правилах Портера–Дафа можно прочитать в книге Джеймса Фоли (James D. Foley), Андриса ван Дама (Andries van Dam), Стивена Фэйнера (Steven K. Feiner) и др. под названием *Computer Graphics: Principles and Practice, Second Edititon in C.*

Таблица 7.1. Правила композиции Портера–Даффа

Правило	Описание
`CLEAR`	Изображение-источник удаляет пиксели целевого изображения
`SRC`	Изображение-источник перезаписывает пиксели целевого изображения и пустые пиксели
`DST`	Изображение-источник не влияет на целевое изображение
`SRC_OVER`	Изображение-источник смешивается с целевым изображением и перезаписывает пустые пиксели
`DST_OVER`	Изображение-источник не влияет на целевое изображение и перезаписывает пустые пиксели
`SRC_IN`	Изображение-источник перезаписывает только пиксели целевого изображения
`SRC_OUT`	Изображение-источник удаляет пиксели целевого изображения и перезаписывает пустые пиксели
`DST_IN`	Альфа-значение источника модифицирует пиксели целевого изображения
`DST_OUT`	Дополнение до альфа-значения источника модифицирует пиксели целевого изображения
`SRC_ATOP`	Часть изображения-источника, лежащая внутри целевого изображения, смешивается с целевым изображением
`DST_ATOP`	Изображение-источник смешивается с целевым изображением
`XOR`	Дополнение до альфа-значения источника модифицирует целевое изображение. Пустые пиксели перезаписываются

Для создания объекта-правила композиции, который реализует интерфейс `Composite`, используется метод `setComposite()` класса `Graphics2D`. В интерфейсе Java 2D API содержится только один такой класс, а именно — `AlphaComposite`, который реализует восемь правил Портера–Дафа (рис. 7.23).

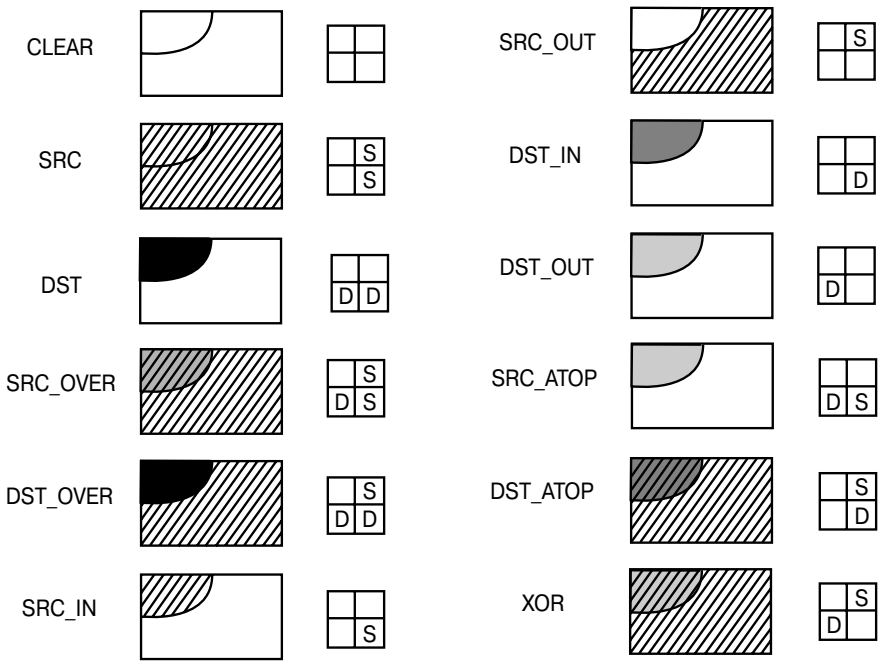


Рис. 7.23. Правила композиции Портера–Дафа

Для создания экземпляра правила типа `AlphaComposite` нужно указать для метода `getInstance()` этого класса правило и альфа-значение для пикселей изображения-источника:

```
int rule = AlphaComposite.SRC_OVER;
float alpha = 0.5f;
g2.setComposite(AlphaComposite.getInstance(rule, alpha));
g2.setPaint(Color.blue);
g2.fill(rectangle);
```

В данном фрагменте кода прямоугольник заполняется синим цветом и для него задается альфа-значение `0.5`. Поскольку установлено правило композиции `SRC_OVER`, его пиксели смешиваются с учетом коэффициента прозрачности с уже существующим изображением.

Программа, код которой приведен в листинге 7.3, позволяет исследовать правила композиции. Выберите правило в раскрывающемся списке и используйте линейный регулятор, чтобы установить альфа-значение для прямоугольника, который является изображением-источником типа `AlphaComposite`.

В нижней части окна программы отображается описание каждого правила. Обратите внимание, что это описание автоматически вычисляется на основе диаграммы правил композиции. Например, строка `DS` во втором ряду диаграммы приводит к включению в результирующее описание строки “blends with destination” (“смешивается с целевым изображением”).

Однако эта программа не гарантирует, что используемый графический контекст имеет альфа-канал. Когда пиксели накладываются на целевое изображение без альфа-канала, то цвета пикселей умножаются на альфа-значение, и альфа-значение отбрасывается. Поскольку несколько правил Портера–Дафа используют альфа-значения для целевого изображения, наличие этого значения играет важную роль. Поэтому для композиции данных фигур используется буферное изображение с цветовой моделью ARGB. После объединения рисунков результат отображается на экране.

```
BufferedImage image = new BufferedImage(getWidth(),
   getHeight(),  BufferedImage.TYPE_INT_ARGB);
Graphics2D gImage = image.createGraphics();
// Рисование с помощью gImage.
g2.drawImage(image, null, 0, 0);
```

Полный код этой программы приведен в листинге 7.3, а на рис. 7.24 показан внешний вид ее окна. После запуска программы переместите ползунок регулятора слева направо для того, чтобы посмотреть результат применения выбранного правила композиции фигур для разных альфа-значений. Обратите особое внимание на то, что единственным различием между правилами `DST_IN` и `DST_OUT` является направление изменения цвета целевого изображения (!) при изменении альфа-значения изображения-источника.

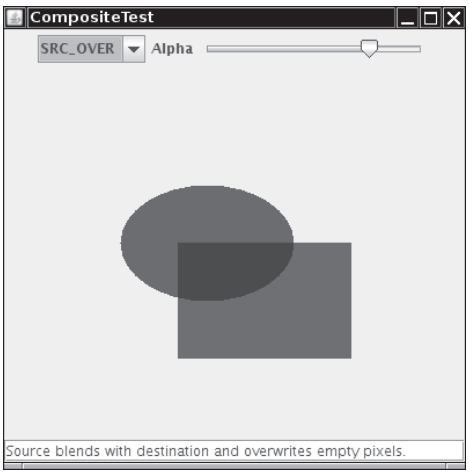


Рис. 7.24. Результат работы программы `CompositeTest`

Листинг 7.3. Содержимое файла `CompositeTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.awt.image.*;
4.  import java.awt.geom.*;
5.  import javax.swing.*;
6.  import javax.swing.event.*;
7.
8.  /**
9.   * Демонстрация правил Портера-Даффа
10.  * для объединения изображений.
11.  * @version 1.03 2007-08-16
12.  * @author Cay Horstmann
13.  */
```

```
14. public class CompositeTest
15. {
16.    public static void main(String[] args)
17.    {
18.       EventQueue.invokeLater(new Runnable()
19.        {
20.          public void run()
21.          {
22.            JFrame frame = new CompositeTestFrame();
23.            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24.            frame.setVisible(true);
25.          }
26.        });
27.    }
28. }
29.
30. /**
31.  * Фрейм с раскрывающимся списком для выбора правила композиции,
32.  * линейным регулятором для изменения альфа-значения, а также
33.  * компонентом для отображения результата применения
34.  * выбранного правила.
35.  */
36. class CompositeTestFrame extends JFrame
37. {
38.    public CompositeTestFrame()
39.    {
40.       setTitle("CompositeTest");
41.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
42.
43.       canvas = new CompositeComponent();
44.       add(canvas, BorderLayout.CENTER);
45.
46.       ruleCombo = new JComboBox(new  Object[] {
47.            new Rule("CLEAR", "  ", "  "),
48.            new Rule("SRC", " S", " S"),
49.            new Rule("DST", "  ", "DD"),
50.            new Rule("SRC_OVER", " S", "DS"),
51.            new Rule("DST_OVER", " S", "DD"),
52.            new Rule("SRC_IN", "  ", " S"),
53.            new Rule("SRC_OUT", " S", "  "),
54.            new Rule("DST_IN", "  ", " D"),
55.            new Rule("DST_OUT", "  ", "D "),
56.            new Rule("SRC_ATOP", "  ", "DS"),
57.            new Rule("DST_ATOP", " S", " D"),
58.            new Rule("XOR", " S", "D "),});
59.       ruleCombo.addActionListener(new ActionListener()
60.          {
61.             public void actionPerformed(ActionEvent event)
62.             {
63.                Rule r = (Rule)ruleCombo.getSelectedItem();
64.                canvas.setRule(r.getValue());
65.                explanation.setText(r.getExplanation());
66.             }
67.          });
68.
69.       alphaSlider = new JSlider(0, 100, 75);
70.       alphaSlider.addChangeListener(new
```

```
71.         ChangeListener()
72.         {
73.            public void stateChanged(ChangeEvent event)
74.            {
75.               canvas.setAlpha(alphaSlider.getValue());
76.            }
77.         });
78.       JPanel panel = new JPanel();
79.       panel.add(ruleCombo);
80.       panel.add(new JLabel("Alpha"));
81.       panel.add(alphaSlider);
82.       add(panel, BorderLayout.NORTH);
83.
84.       explanation = new JTextField();
85.       add(explanation, BorderLayout.SOUTH);
86.
87.       canvas.setAlpha(alphaSlider.getValue());
88.       Rule r = (Rule) ruleCombo.getSelectedItem();
89.       canvas.setRule(r.getValue());
90.       explanation.setText(r.getExplanation());
91.    }
92.
93.    private CompositePanel canvas;
94.    private JComboBox ruleCombo;
95.    private JSlider alphaSlider;
96.    private JTextField explanation;
97.    private static final int DEFAULT_WIDTH = 400;
98.    private static final int DEFAULT_HEIGHT = 400;
99. }
100.
101. /**
102.  * Класс, описывающий правило Портера-Даффа.
103.  */
104. class Rule
105. {
106.    /**
107.     *  Создание правила Портера-Даффа.
108.     *  @param n Имя правила
109.     *  @param pd1 Первая строка квадрата Портера-Даффа
110.     *  @param pd2 Вторая строка квадрата Портера-Даффа
111.     */
112.    public Rule(String n, String pd1, String pd2)
113.    {
114.       name = n;
115.       porterDuff1 = pd1;
116.       porterDuff2 = pd2;
117.    }
118.
119.    /**
120.     * Метод, получающий объяснение поведения данного правила.
121.     * @return Объяснение
122.     */
123.    public String getExplanation()
124.    {
125.       StringBuilder r = new StringBuilder("Source ");
126.       if (porterDuff2.equals("  ")) r.append("clears");
127.       if (porterDuff2.equals(" S")) r.append("overwrites");
```

```
128.      if (porterDuff2.equals("DS")) r.append("blends with");
129.      if (porterDuff2.equals(" D")) r.append("alpha modifies");
130.      if (porterDuff2.equals("D "))
131.        r.append("alpha complement modifies");
132.      if (porterDuff2.equals("DD")) r.append("does not affect");
133.      r.append(" destination");
134.      if (porterDuff1.equals(" S"))
135.        r.append(" and overwrites empty pixels");
136.      r.append(".");
137.      return r.toString();
138.    }
139.
140.    public String toString()
141.    {
142.      return name;
143.     }
144.
145.    /**
146.     *  Метод, получающий значение данного правила в классе AlphaComposite.
147.     *  @return Константа класса AlphaComposite или -1,
148.     *  если подходящей константы нет.
149.     */
150.    public int getValue()
151.    {
152.      try
153.      {
154.        return (Integer)
155.          AlphaComposite.class.getField(name).get(null);
156.      }
157.      catch (Exception e)
158.      {
159.        return -1;
160.      }
161.    }
162.
163.    private String name;
164.    private String porterDuff1;
165.    private String porterDuff2;
166. }
167.
168. /**
169.  * Компонент, выполняющий рисование двух фигур и объединяющий
170.  * их с учетом правил композиции.
171.  */
172. class CompositeComponent extends JComponent
173. {
174.    public CompositeComponent()
175.    {
176.      shape1 = new Ellipse2D.Double(100, 100, 150, 100);
177.      shape2 = new Rectangle2D.Double(150, 150, 150, 100);
178.    }
179.
180.    public void paintComponent(Graphics g)
181.    {
182.      super.paintComponent(g);
183.      Graphics2D g2 = (Graphics2D)g;
184.
```

```
185.      BufferedImage image = new BufferedImage(getWidth(), getHeight(),
186.                                    BufferedImage.TYPE_INT_ARGB);
187.      Graphics2D gImage = image.createGraphics();
188.      gImage.setPaint(Color.red);
189.      gImage.fill(shape1);
190.      AlphaComposite composite = AlphaComposite.getInstance(rule, alpha);
191.      gImage.setComposite(composite);
192.      gImage.setPaint(Color.blue);
193.      gImage.fill(shape2);
194.      g2.drawImage(image, null, 0, 0);
195.   }
196.
197.   /**
198.    * Установка правила композиции.
199.    * @param r Правило (в виде константы AlphaComposite)
200.    */
201.   public void setRule(int r)
202.   {
203.      rule = r;
204.      repaint();
205.   }
206.
207.   /**
208.    * Установка значения альфа источника.
209.    * @param a Значение альфа в интервале от 0 до 100
210.    */
211.   public void setAlpha(int a)
212.   {
213.      alpha = (float) a / 100.0F;
214.      repaint();
215.   }
216.
217.   private int rule;
218.   private Shape shape1;
219.   private Shape shape2;
220.   private float alpha;
221. }
```

API java.awt.Graphics2D 1.2

- `void setComposite(Composite s)`

  На основе заданного объекта, который реализует интерфейс `Composite`, создает композицию для данного графического контекста.

API java.awt.AlphaComposite 1.2

- `static AlphaComposite getInstance(int rule)`
- `static AlphaComposite getInstance(int rule, float sourceAlpha)`

  На основе заданного правила и значения альфа создает объект композиции. Для параметра `rule` может указываться одно из следующих значений: `CLEAR`, `DST`, `SRC`, `SRC_OVER`, `DST_OVER`, `SRC_ATOP`, `SRC_IN`, `SRC_OUT`, `DST_ATOP`, `DST_IN`, `DST_OUT`, `XOR`.

Параметры рисования

В предыдущих разделах можно было увидеть, что процесс визуализации является довольно сложным. И хотя чаще всего Java 2D API работает удивительно быстро, в некоторых случаях бывает необходимо иметь контроль над процессом выбора компромисса между скоростью и качеством. Получать такой контроль можно путем задания так называемых *рекомендаций по визуализации* (rendering hints), здесь и далее именуемых просто *параметрами рисования*. Метод `setRenderingHint()` класса `Graphics2D` позволяет задавать лишь один такой параметр. А вообще ключи и значения таких параметров объявляются в классе `RenderingHints`. В табл. 7.2 вкратце перечислены основные возможные варианты. Значения, оканчивающиеся конструкцией `_DEFAULT`, обозначают подлежащие использованию по умолчанию варианты, которые в конкретно данной реализации гарантируют достижение наилучшего компромисса между производительностью и качеством.

Таблица 7.2. Параметры рисования

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 для принятия решения о том, следует ли выполнять его сглаживание в случае такого размера, а значения со словом `LCD` — визуализацию подпикселей для мониторов соответствующего типа
`KEY_FRACTIONAL_METRICS`	`VALUE_FRACTIONALMETRICS_ON` `VALUE_FRACTIONALMETRICS_OFF` `VALUE_FRACTIONALMETRICS_DEFAULT`	Включение/отключение вычисления дробных размеров шрифтов. Использование дробных размеров шрифтов позволяет улучшить их расположение
`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 того, должно ли размещение штрихов контролироваться графическим ускорителем (что чревато их смещением на целую половину пикселя) или вычисляться строго по правилу, которое требует, чтобы штрихи обязательно 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 расположенных рядом пикселей. (Обратите внимание на то, 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 композиции на основе альфа-значений
KEY_COLOR_RENDERING	VALUE_COLOR_RENDER_QUALITY VALUE_COLOR_RENDER_SPEED VALUE_COLOR_RENDER_DEFAULT	Выбор качества или скорости при отображении цветов. Такой выбор необходимо делать только в том случае, когда используются разные цветовые пространства
KEY_INTERPOLATION	VALUE_INTERPOLATION_NEAREST_NEIGHBOR VALUE_INTERPOLATION_BILINEAR VALUE_INTERPOLATION_BICUBIC	Выбор правила интерполяции пикселей при масштабировании рисунка

Наиболее полезным из этих параметров является *сглаживание* (antialiasing), подразумевающее удаление всевозможных неровностей при рисовании различных линий. Как показано на рис. 7.25, линия рисуется в виде "лестницы" из расположенных уступом пикселей. Иногда, особенно на мониторах с низким разрешением, она выглядит совершенно неприемлемо. Но если вместо рисования пикселей кривой только двумя контрастными цветами (например, черным и белым) нарисовать ее полутонами, значения которых пропорциональны занимаемой площади пикселя, то такая линия будет выглядеть более гладкой. Этот метод рисования называется сглаживанием. Конечно, он требует несколько больше времени, которое расходуется на расчет значений переходных цветов.

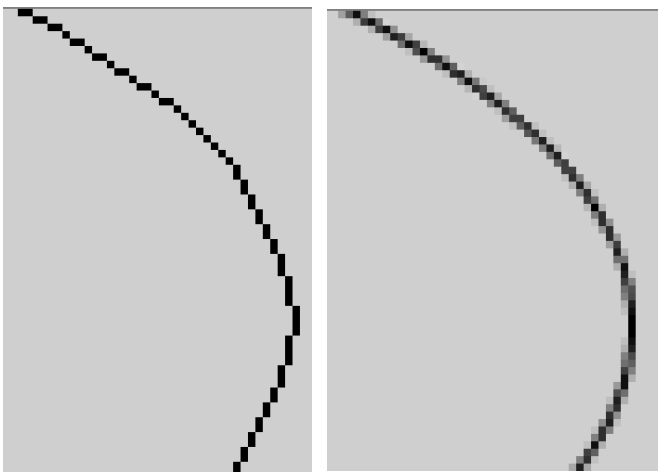


Рис. 7.25. Сглаживание

Итак, для включения сглаживания фигур нужно использовать следующий метод:

```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
  RenderingHints.VALUE_ANTIALIAS_ON);
```

А для включения сглаживания шрифтов используется приведенный ниже вызов:

```
g2.setRenderingHint (RenderingHints.KEY_TEXT_ANTIALIASING,
  RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
```

Остальные параметры рисования используются гораздо реже.

Пары "ключ–значение" параметра рисования можно сгруппировать в объекте-карте и задавать их одновременно с помощью метода `setRenderingHints()`. Для этого подойдет любой класс, реализующий интерфейс `Map`, а значит, и сам класс `RenderingHints`. Он также реализует интерфейс `Map` и формирует карту по умолчанию, если параметр его конструктора имеет значение `null`.

```
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.setRenderingHint(hints);
```

Применение такого метода демонстрирует программа, код которой приведен в листинге 7.4. В ней используется несколько параметров рисования, которые показались нам наиболее полезными. Ниже перечислены моменты, на которые требуется обратить особое внимание.

- Под сглаживанием подразумевается устранение неровностей в эллипсе.
- Под текстовым сглаживанием подразумевается устранение неровностей в тексте.
- На некоторых платформах из-за дробной системы метрики текста буквы будут сдвигаться немного ближе друг к другу.
- Выбор значения `VALUE_RENDER_QUALITY` подразумевает устранение неровностей в масштабируемом изображении. (Точно такого же эффекта можно было бы добиться и путем установки для `KEY_INTERPOLATION` значения `VALUE_INTERPOLATION_BICUBIC`.)
- 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 снимок экрана этой программы.

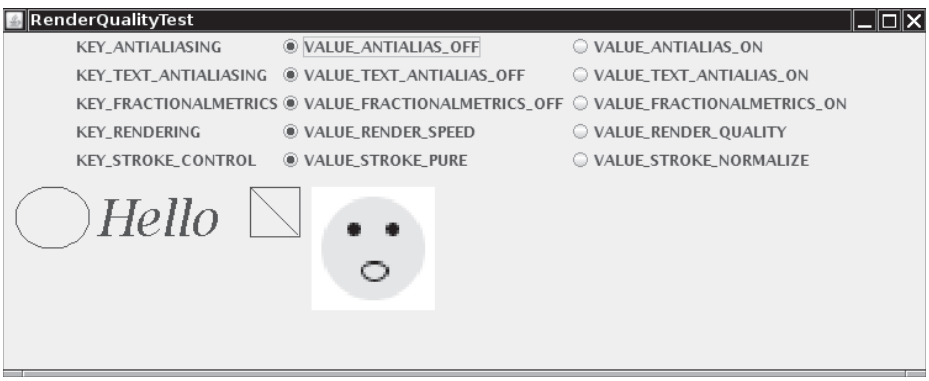


Рис. 7.26. Тестирование действия параметров рисования

Листинг 7.4. Содержимое файла `RenderQualityTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.awt.geom.*;
4.  import java.io.*;
5.  import javax.imageio.*;
6.  import javax.swing.*;
7.  /**
8.   * Программа, демонстрирующая действие
9.   * различных параметров рисования.
10.  * @version 1.10 2007-08-16
11.  * @author Cay Horstmann
12.  */
13. public class RenderQualityTest
14. {
15.   public static void main(String[] args)
16.   {
17.     EventQueue.invokeLater(new Runnable()
18.      {
19.        public void run()
20.        {
21.          JFrame frame = new RenderQualityTestFrame();
22.          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.          frame.setVisible(true);
24.        }
25.     });
26.   }
27. }
28. /**
29.  * Фрейм, содержащий переключатели для указания
30.  * параметров рисования и изображение, которое
31.  * рисуется с их использованием.
32.  */
33. class RenderQualityTestFrame extends JFrame
34. {
35.   public RenderQualityTestFrame()
36.   {
37.      setTitle("RenderQualityTest");
38.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
39.
40.      buttonBox = new JPanel();
41.      buttonBox.setLayout(new GridBagLayout());
42.      hints = new RenderingHints(null);
43.
44.      makeButtons("KEY_ANTIALIASING", "VALUE_ANTIALIAS_OFF", "VALUE_ANTIALIAS_ON");
45.      makeButtons("KEY_TEXT_ANTIALIASING", "VALUE_TEXT_ANTIALIAS_OFF",
46.                  "VALUE_TEXT_ANTIALIAS_ON");
47.      makeButtons("KEY_FRACTIONALMETRICS", "VALUE_FRACTIONALMETRICS_OFF",
48.                   "VALUE_FRACTIONALMETRICS_ON");
49.      makeButtons("KEY_RENDERING", "VALUE_RENDER_SPEED", "VALUE_RENDER_QUALITY");
50.      makeButtons("KEY_STROKE_CONTROL", "VALUE_STROKE_PURE", "VALUE_STROKE_NORMALIZE");
51.      canvas = new RenderQualityComponent();
52.      canvas.setRenderingHints(hints);
53.      add(canvas, BorderLayout.CENTER);
54.      add(buttonBox, BorderLayout.NORTH);
55.   }
```

```
56.    /**
57.     * Создание набора кнопок для формирования ключа и значений.
58.     * @param key Имя ключа
59.     * @param value1 Имя первого значения ключа
60.     * @param value2 Имя второго значения ключа
61.     */
62.    void makeButtons(String key, String value1, String value2)
63.    {
64.       try
65.       {
66.          final RenderingHints.Key k = (RenderingHints.Key)
67.             RenderingHints.class.getField(key).get(null);
68.          final Object v1 =
69.             RenderingHints.class.getField(value1).get(null);
70.          final Object v2 =
71.             RenderingHints.class.getField(value2).get(null);
72.          JLabel label = new JLabel(key);
73.
74.          buttonBox.add(label, new GBC(0, r).setAnchor(GBC.WEST));
75.          ButtonGroup group = new ButtonGroup();
76.          JRadioButton b1 = new JRadioButton(value1, true);
77.
78.          buttonBox.add(b1, new GBC(1, r).setAnchor(GBC.WEST));
79.          group.add(b1);
80.          b1.addActionListener(new
81.             ActionListener()
82.             {
83.                public void actionPerformed(ActionEvent event)
84.                {
85.                   hints.put(k, v1);
86.                   canvas.setRenderingHints(hints);
87.                }
88.             });
89.          JRadioButton b2 = new JRadioButton(value2, false);
90.
91.          buttonBox.add(b2, new GBC(2, r).setAnchor(GBC.WEST));
92.          group.add(b2);
93.          b2.addActionListener(new ActionListener()
94.             {
95.                public void actionPerformed(ActionEvent event)
96.                {
97.                   hints.put(k, v2);
98.                   canvas.setRenderingHints(hints);
99.                }
100.         });
101.         hints.put(k, v1);
102.         r++;
103.      }
104.      catch (Exception e)
105.      {
106.         e.printStackTrace();
107.      }
108.   }
109.   private RenderQualityPanel canvas;
110.   private JPanel buttonBox;
111.   private RenderingHints hints;
112.   private int r;
```

```
113.   private static final int DEFAULT_WIDTH = 750;
114.   private static final int DEFAULT_HEIGHT = 300;
115. }
116. /**
117.  * Компонент, в котором отображается рисунок, получающийся
118.  * в результате применения параметров рисования.
119.  */
120. class RenderQualityComponent extends JComponent
121. {
122.   public RenderQualityComponent()
123.   {
124.     try
125.     {
126.       image = ImageIO.read(new File("face.gif"));
127.     }
128.     catch (IOException e)
129.     {
130.       e.printStackTrace();
131.     }
132.   }
133.
134.   public void paintComponent(Graphics g)
135.   {
136.     Graphics2D g2 = (Graphics2D) g;
137.     g2.setRenderingHints(hints);
138.     g2.draw(new Ellipse2D.Double(10, 10, 60, 50));
139.     g2.setFont(new Font("Serif", Font.ITALIC, 40));
140.     g2.drawString("Hello", 75, 50);
141.
142.     g2.draw(new Rectangle2D.Double(200, 10, 40, 40));
143.     g2.draw(new Line2D.Double(201, 11, 239, 49));
144.
145.     g2.drawImage(image, 250, 10, 100, 100, null);
146.   }
147.   /**
148.    * Установка параметров и перерисовка изображения.
149.    * @param h Параметры рисования
150.    */
151.   public void setRenderingHints(RenderingHints h)
152.   {
153.     hints = h;
154.     repaint();
155.   }
156.   private RenderingHints hints = new RenderingHints(null);
157.   private Image image;
158. }
```

API java.awt.Graphics2D 1.2

- `void setRenderingHint(RenderingHints.Key key, Object value)`
  Устанавливает параметры рисования для графического контекста.
- `void setRenderingHints(Map m)`
  Устанавливает параметры рисования, пары "ключ–значение" которых хранятся в карте `m`.

API java.awt.RenderingHints 1.2

- `RenderingHints(Map<RenderingHints.Key, ?> m)`

  Создает карту для хранения параметров рисования. Если параметр `m` имеет значение `null`, то реализуется карта по умолчанию.

Чтение и запись изображений

До выхода версии 1.4 возможности для чтения и записи файлов с изображениями в Java SE были весьма ограниченными. Например, считывать можно было только файлы формата GIF и JPEG, а записывать официально вообще нельзя было никаких графических файлов.

Теперь ситуация выглядит значительно лучше. Еще в Java SE 1.4 появился пакет `javax.imageio` с готовыми средствами для чтения и записи нескольких наиболее типичных форматов файлов, а также средой для разработки своих собственных средств для чтения и записи файлов других форматов. Что касается версии Java SE 6, то в ней поддерживаются такие форматы файлов, 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 возможность записи файлов формата GIF не поддерживалась.

Основные предлагаемые в библиотеке возможности выглядят чрезвычайно просто. Для загрузки рисунка применяется статический метод `read()` класса `ImageIO`:

```
File f = ...;
BufferedImage image = ImageIO.read(f);
```

Класс `ImageIO` выбирает соответствующую программу чтения на основе типа файла. Он проверяет расширение файла и соответствующее значение в заголовке файла. Если для чтения данного файла нельзя найти подходящей программы или она не может расшифровать содержимое, то метод `read()` возвращает значение `null`.

Запись рисунка в файл осуществляется так же просто, как и чтение:

```
File f = ...;
String format = ...;
ImageIO.write(image, format, f);
```

Строка `format` определяет формат рисунка (например, `JPEG` или `PNG`). Класс `ImageIO` выбирает соответствующую программу записи и сохраняет рисунок.

Получение объектов для чтения и записи графических файлов

Для выполнения расширенных операций записи и чтения изображений, которые выходят за пределы простого использования статических методов `read()` и `write()` класса `ImageIO`, необходимо прежде получить соответствующие объекты `ImageReader` и `ImageWriter`. В случае если обнаружено соответствие одному из приведенных ниже условий, класс `ImageIO` позволяет организовать перебор всех средств чтения и записи графических файлов:

- формат изображения, например `JPEG`;
- расширение файла, например `jpg`;
- MIME-тип, например `image/jpeg`.



Стандарт MIME (Multipurpose Internet Mail Extensions) определяет общие форматы данных (например, “image/jpeg”, “application/pdf”). HTML-версию документа RFC (Requests for Comments) с описанием стандартов MIME можно найти по адресу:

`http://www.oac.uci.edu/indiv/ehood/MIME`.

Например, с помощью приведенного ниже фрагмента кода можно получить объект для чтения JPEG-файлов:

```
ImageReader reader = null;
Iterator<ImageReader> iter =
   ImageIO.getImageReadersByFormatName("JPEG");
if (iter.hasNext()) reader = iter.next();
```

Методы `getImageReaderBySuffix()` и `getImageReaderByMIMEType()` возвращают объекты чтения графических файлов с указанным расширением или MIME-типом.

Класс `ImageIO` может найти несколько объектов чтения, каждый из которых способен обработать данный тип файла. В этом случае следует выбрать один из объектов на основе более подробной информации, которую можно получить с помощью *интерфейса поставщика услуг* (service provider interface):

```
ImageReaderSpi spi = reader.getOriginatingProvider();
```

Затем можно получить сведения об изготовителе и номер версии:

```
String vendor = spi.getVendor();
String version = spi.getVersion();
```

Возможно, эта информация поможет сделать выбор нужного объекта чтения или предоставит возможность создать список доступных средств, для того чтобы пользователи могли выбрать наиболее подходящее из них по своему усмотрению. Однако давайте пока считать, что нам подходит первый объект из этого списка.

В примере программы, код которой приведен в листинге 7.5, предполагается, что требуется найти все суффиксы файлов, которые применяются всеми доступными средствами чтения, дабы потом использовать их в файловом фильтре. В случае версии Java SE 6 для этой цели можно применить статический метод `ImageIO.getReaderFileSuffixes`:

```
String[] extensions = ImageIO.getWriterFileSuffixes();
chooser.setFileFilter(new FileNameExtensionFilter("Image files", extensions));
```

Что касается сохранения файлов, то здесь требуется приложить больше усилий. Нам бы хотелось, чтобы пользователю отображалось меню со всеми поддерживаемыми форматами изображений. Метод `getWriterFormaNames()` класса `IOImage` для этой цели, к сожалению, не подходит, потому что он будет возвращать довольно необычный перечень с лишними вариантами названий форматов, например, такой:

```
jpg, BMP, bmp, JPG, jpeg, wbmp, png, JPEG, PNG, WBMP, GIF, gif
```

Это не совсем то, что должно отображаться в нашем меню. Нам необходимо, чтобы это был перечень только “предпочитаемых” вариантов названий форматов. Поэтому мы используем для этой цели вспомогательный метод `getWriterFormats` (см. листинг 7.5). Мы сначала отыскиваем первый объект записи, который ассоциируется с названием каждого формата, а затем спрашиваем его, какие названия форматов у него имеются, с надеждой, что первым он отобразит именно наиболее популярный вариант. В случае объекта, отвечающего за запись JPEG-файлов, этот подход будет работать отлично, и приводить к отображению перед всеми остальными вариантами именно варианта `JPEG`. (В случае объекта, отвечающего за запись PNG-файлов, одна-

ко, первым будет отображаться не вариант PNG, а его прописной эквивалент png. Мы очень надеемся, что в ближайшем будущем эта проблема будет устранена, а пока что справляемся с ней путем принудительного преобразования всех прописных букв в названии форматов в заглавные.) Выбрав предпочитаемый вариант, мы просто удаляем все альтернативные названия из исходного набора и делаем это до тех пор, пока не обработаем все названия форматов.

Чтение и запись файлов, содержащих несколько изображений

Некоторые файлы, например анимационные GIF-файлы, могут содержать несколько изображений. Однако метод read() класса ImageIO позволяет считывать только одно из них. Для чтения нескольких изображений нужно преобразовать источник входных данных (например, входной поток файла) в объект ImageInputStream:

```
InputStream in = ...;
ImageInputStream imageIn = ImageIO.createImageInputStream(in);
```

Затем следует соединить этот объект с программой чтения:

```
reader.setInput(imageIn, true);
```

Второй параметр имеет значение true, а это означает, что входной поток позволяет читать данные только в прямом направлении. Значение false допускает произвольный доступ, обеспечиваемый либо за счет буферизации, либо путем организации произвольного доступа к файлу. Произвольный доступ позволяет эффективно выполнять определенные операции. Например, для подсчета количества изображений в GIF-файле нужно прочитать полностью весь файл. Если затем необходимо выбрать какой-то рисунок, то в режиме произвольного доступа не придется снова считывать весь входной поток.

Сказанное важно только для случая чтения входного потока, содержащего несколько изображений, и при отсутствии в заголовке файла требуемой информации о формате рисунка (например, его номера). Для чтения файла можно применить показанный ниже простой фрагмент кода:

```
File f = ...;
ImageInputStream imageIn = ImageIO.createImageInputStream(f);
Reader.setInput(imageIn);
```

Таким образом, при наличии объекта чтения можно считывать отдельные рисунки по известному индексу, отсчет которого начинается с нуля:

```
BufferedImage image = reader.read(index);
```

Если для потока на входе установлен режим чтения “только вперед”, то следует продолжать считывать рисунки до тех пор, пока метод read() не сгенерирует исключение IndexOutOgBoundsException. Для этого можно также вызвать метод getNumImages(), параметр которого имеет значение true, т.е. разрешается поиск и подсчет количества изображений:

```
int n = reader.getNumImages(true);
```

Если для входного потока задан режим просмотра “только вперед”, данный метод генерирует исключение IllegalStateException. Кроме того, в качестве параметра метода getNumImages() может быть задано значение false. Если метод getNumImages() не может определить количество рисунков без проведения поиска, то он возвращает значение -1. В этом случае нужно считывать изображения до появления исключения IndexOutOfBoundsException.

Некоторые файлы могут содержать миниатюры (thumbnail), представляющие собой уменьшенные версии основного изображения, предназначенные для предварительного просмотра. С помощью следующего вызова можно узнать количество миниатюр для изображения:

```
int count = reader.getNumThumbnails(index);
```

Для получения миниатюры можно применить метод `getThumbnail()` с указанием индекса основного изображения и индекса самой миниатюры:

```
BufferedImage thumbnail = reader.getThumbnail(index, thumbnailIndex);
```

Иногда необходимо получить размер рисунка еще до получения самого рисунка. Это особенно важно, когда речь идет о передаче рисунка очень большого размера или использовании очень низкоскоростного сетевого соединения. Для получения размеров рисунка с заданным индексом применяются следующие методы:

```
int width = reader.getWidth(index);
int height = reader.getHeight(index);
```

Для записи файла с несколькими рисунками нужно сначала создать объект `ImageWriter`, а затем с помощью класса `IOImage` перечислить все объекты, способные записывать изображения данного формата:

```
String format = . . .;
ImageWriter writer = null;
Iterator<ImageWriter> iter =
   ImageIO.getImageWritersByFormatName( format );
if (iter.hasNext()) writer = iter.next();
```

Затем следует преобразовать выходной поток в объект `ImageOutputStream` и подключить его к объекту записи:

```
File f =...;
ImageOutputStream imageOut = ImageIO.createImageOutputStream(f);
writer.setOutput(imageOut);
```

Каждое изображение нужно оформить в виде объекта `IIOImage`, затем (по желанию) составить список миниатюр и метаданные (например, алгоритм сжатия и информацию о цвете). В рассматриваемом здесь примере для списка миниатюр и метаданных заданы значения `null`. Более подробную информацию по данному вопросу можно найти в документации по API.

```
IIOImage iioImage = new IIOImage(images[i], null, null);
```

Для записи *первого* изображения используется метод `write()`:

```
writer.write(new IIOImage(images[0], null, null));
```

Для всех последующих изображений выполняется проверка:

```
if (writer.canInsertImage(i))
   writer.writeInsert(i, iioImage, null);
```

Третий параметр может содержать объект `ImageWriteParam`, позволяющий указать такие подробности, как мозаичное расположение и сжатие. В этом примере в качестве третьего параметра используется значение `null`.

Не все форматы файлов могут содержать несколько изображений. В таком случае метод `canInsertImage()` возвращает для `i > 0` значение `false` и сохраняется только один рисунок.

Программа, код которой приведен в листинге 7.5, позволяет загружать и сохранять файлы в форматах, для которых в библиотеке Java предусмотрены средства чтения и записи. Она может отображать сразу несколько изображений (рис. 7.27), но не миниатюры.

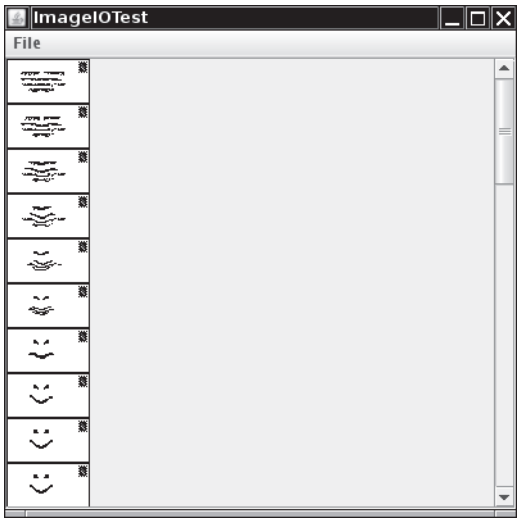


Рис. 7.27. Анимированное GIF-изображение

Листинг 7.5. Содержимое файла `ImageIOTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.awt.image.*;
4.  import java.io.*;
5.  import java.util.*;
6.  import javax.imageio.*;
7.  import javax.imageio.stream.*;
8.  import javax.swing.*;
9.  import javax.swing.filechooser.*;
10.
11. /**
12.  * Данная программа позволяет читать и записывать файлы изображений
13.  * в тех форматах, которые поддерживаются JDK, а также умеет
14.  * работать и с такими файлами, в которых содержится по нескольку
15.  * изображений.
16.  * @version 1.02 2007-08-16
17.  * @author Cay Horstmann
18.  */
19. public class ImageIOTest
20. {
21.    public static void main(String[] args)
22.    {
23.      EventQueue.invokeLater(new Runnable()
24.       {
25.         public void run()
26.         {
27.           JFrame frame = new ImageIOFrame();
28.           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29.           frame.setVisible(true);
30.
31.       });
32.    }
```

```
33.  }
34.
35.  /**
36.   *  Фрейм для отображения загружаемых изображений и меню
37.   *  с пунктами для загрузки и сохранения файлов.
38.   */
39.  class ImageIOFrame extends JFrame
40.  {
41.    public ImageIOFrame()
42.    {
43.      setTitle("ImageIOTest");
44.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
45.
46.      JMenu fileMenu = new JMenu("File");
47.      JMenuItem openItem = new JMenuItem("Open");
48.      openItem.addActionListener(new ActionListener()
49.        {
50.          public void actionPerformed(ActionEvent event)
51.          {
52.            openFile();
53.          }
54.        });
55.      fileMenu.add(openItem);
56.
57.      JMenu saveMenu = new JMenu("Save");
58.      fileMenu.add(saveMenu);
59.      Iterator<String> iter = writerFormats.iterator();
60.      while (iter.hasNext())
61.      {
62.        final String formatName = iter.next();
63.        JMenuItem formatItem = new JMenuItem(formatName);
64.        saveMenu.add(formatItem);
65.        formatItem.addActionListener(new ActionListener()
66.          {
67.            public void actionPerformed(ActionEvent event)
68.            {
69.              saveFile(formatName);
70.            }
71.          });
72.      }
73.
74.      JMenuItem exitItem = new JMenuItem("Exit");
75.      exitItem.addActionListener(new ActionListener()
76.        {
77.          public void actionPerformed(ActionEvent event)
78.          {
79.            System.exit(0);
80.          }
81.        });
82.      fileMenu.add(exitItem);
83.
84.
85.      JMenuBar menuBar = new JMenuBar();
86.      menuBar.add(fileMenu);
87.      setJMenuBar(menuBar);
88.    }
89.
```

```
90.    /**
91.     * Открытие файла и загрузка изображений.
92.     */
93.    public void openFile()
94.    {
95.       JFileChooser chooser = new JFileChooser();
96.       chooser.setCurrentDirectory(new File("."));
97.       String[] extensions = ImageIO.getReaderFileSuffixes();
98.       chooser.setFileFilter(new FileNameExtensionFilter("Image files", ex tensions));
99.       int r = chooser.showOpenDialog(this);
100.      if (r != JFileChooser.APPROVE_OPTION) return;
101.      File f = chooser.getSelectedFile();
102.      Box box = Box.createVerticalBox();
103.      try
104.      {
105.         String name = f.getName();
106.         String suffix =
107.            name.substring(name.lastIndexOf('.') + 1);
108.         Iterator<ImageReader> iter
109.           = ImageIO.getImageReadersBySuffix(suffix);
110.         ImageReader reader = iter.next();
111.         ImageInputStream imageIn = ImageIO.createImageInputStream(f);
112.         reader.setInput(imageIn);
113.         int count = reader.getNumImages(true);
114.         images = new BufferedImage[count];
115.         for (int i = 0; i < count; i++)
116.         {
117.            images[i] = reader.read(i);
118.            box.add(new JLabel(new ImageIcon(images[i])));
119.         }
120.      }
121.      catch (IOException e)
122.      {
123.         JOptionPane.showMessageDialog(this, e);
124.      }
125.      setContentPane(new JScrollPane(box));
126.      validate();
127.   }
128.
129.   /**
130.    * Сохранение текущего изображения в файле.
131.    * @param formatName Формат файла
132.    */
133.   public void saveFile(final String formatName)
134.   {
135.      if (images == null) return;
136.      Iterator<ImageWriter> iter =
137.        ImageIO.getImageWritersByFormatName(formatName);
138.      ImageWriter writer = iter.next();
139.      JFileChooser chooser = new JFileChooser();
140.      chooser.setCurrentDirectory(new File("."));
141.      String[] extensions = writer.getOriginatingProvider().getFileSuffixes();
142.      chooser.setFileFilter(new FileNameExtensionFilter("Image files", extensions));
143.
144.      int r = chooser.showSaveDialog(this);
145.      if (r != JFileChooser.APPROVE_OPTION) return;
146.      File f = chooser.getSelectedFile();
```

```
147.      try
148.      {
149.         ImageOutputStream imageOut =
150.            ImageIO.createImageOutputStream(f);
151.         writer.setOutput(imageOut);
152.
153.         writer.write(new IIOImage(images[0], null, null));
154.         for (int i = 1; i < images.length; i++)
155.         {
156.            IIOImage iioImage =
157.               new IIOImage(images[i], null, null);
158.            if (writer.canInsertImage(i))
159.               writer.writeInsert(i, iioImage, null);
160.         }
161.      }
162.      catch (IOException e)
163.      {
164.         JOptionPane.showMessageDialog(this, e);
165.      }
166.   }
167.
168.
169.   /**
170.    * Получение "предпочитаемых" названий форматов для всех
171.    * объектов записи. Предпочитаемым является то название,
172.    * которое объект записи отображает первым.
173.    * @return Набор названий форматов
174.    */
175.   public static Set<String> getWriterFormats()
176.   {
177.      TreeSet<String> writerFormats = new TreeSet<String>();
178.      TreeSet<String> formatNames = new
179.         TreeSet<String>(Arrays.asList
180.         (ImageIO.getWriterFormatNames()));
181.      while (formatNames.size() > 0)
182.      {
183.         String name = formatNames.iterator().next();
184.         Iterator<ImageWriter> iter =
185.            ImageIO.getImageWritersByFormatName(name);
186.         ImageWriter writer = iter.next();
187.         String[] names =
188.                 writer.getOriginatingProvider().getFormatNames();
189.         String format = names[0];
190.         if (format.equals(format.toLowerCase())) format =
191.                                    format.toUpperCase();
192.         writerFormats.add(format);
193.         formatNames.removeAll(Arrays.asList(names));
194.      }
195.      return writerFormats;
196.   }
197.
198.   private BufferedImage[] images;
199.   private static Set<String> writerFormats = getWriterFormats();
200.   private static final int DEFAULT_WIDTH = 400;
201.   private static final int DEFAULT_HEIGHT = 400;
202. }
```

API javax.imageio.ImageIO 1.4

- `static BufferedImage read(File input)`
- `static BufferedImage read(InputStream input)`
- `static BufferedImage read(URL input)`

  Считывает изображение из указанного источника.

- `static boolean write(RenderedImage image, String formatName, File output)`
- `static boolean write(RenderedImage image, String formatName, OutputStream output)`

  Записывает изображение в `output`. Возвращает значение `false`, если соответствующий объект 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)`

  Возвращает все объекты чтения или записи, поддерживающие указанный формат, расширение файла или MIME-тип.

- `static String[] getReaderFormatNames()`
- `static String[] getReaderMIMETypes()`
- `static String[] getWriterFormatNames()`
- `static String[] getWriterMIMETypes()`
- `static String[] getReaderFileSuffixes()` 6
- `static String[] getWriterFileSuffixes()` 6

  Получает названия всех форматов и всех типов MIME, а также все файловые суффиксы, которые поддерживаются объектами чтения и записи.

- `ImageInputStream createImageInputStream(Object input)`
- `ImageOutputStream createImageOutputStream(Object output)`

  Создает на базе заданного объекта входной или выходной поток изображения. В качестве параметра может быть указан файл, поток, экземпляр класса `RandomAccessFile` или другой объект, для которого существует соответствующий поставщик услуг. Если для объекта не зарегистрирован поставщик услуг, то в таком случае возвращается значение `null`.

API javax.imageio.ImageReader 1.4

- `void setInput(Object input)`
- `void setInput(Object input, boolean seekForwardOnly)`

  Задает источник входных данных для чтения.

Параметры:	`input`	Экземпляр класса `ImageInputStream` или другой объект, с которым может работать данный объект чтения
	`seekForwardOnly`	Если задано значение `true`, то чтение осуществляется только в прямом направлении; по умолчанию программа чтения использует произвольный доступ и, если нужно, буферизирует данные

- `BufferedImage read(int index)`

  Считывает изображение с заданным индексом (отсчет начинается с нуля). Если такого изображения нет, генерируется исключение `IndexOutOfBoundsException`.
- `int getNumImages(boolean allowSearch)`

  Возвращает количество изображений для объекта чтения. Если параметр `allowSearch` имеет значение `false` и количество изображений не может быть определено без непосредственного чтения, возвращается значение `-1`. Если параметр `allowSearch` имеет значение `true` и программа находится в режиме прямого чтения, генерируется исключение `IllegalStateException`.
- `int getNumThumbnails(int index)`

  Возвращает количество миниатюр для изображения с заданным индексом.
- `BufferedImage readThumbnail(int index, int thumbnailIndex)`

  Возвращает миниатюру с индексом `thumbnailIndex` для изображения с индексом `index`.
- `int getWidth(int index)`
- `int getHeight(int index)`

  Возвращает ширину и высоту изображения. Если изображение не найдено, то генерируется исключение `IndexOutOfBoundsException`.
- `ImageReaderSpi getOriginatingProvider()`

  Возвращает поставщика услуг, который создал объект чтения.

API javax.imageio.spi.IIOServiceProvider 1.4

- `String getVendorName()`
- `String getVersion()`

  Получает имя и версию поставщика данной службы.

API javax.imageio.spi.ImageReaderWriterSpi 1.4

- `String[] getFormatNames()`
- `String[] getFileSuffixes()`
- `String[] getMIMETypes()`

  Получает имена форматов, суффиксы файлов и MIME-типы, которые поддерживаются объектами чтения или записи, созданными поставщиком данной службы.

API javax.imageio.ImageWriter 1.4

- `void setOutput(Object output)`

  Устанавливает целевой объект для записи.

  Параметр: `input` Экземпляр класса `ImageOutputStream` или другой объект, с которым может работать объект записи

- `void write(IIOImage image)`
- `void write(RenderedImage image)`

  Записывает единичное изображение в выходной поток.

- `void writeInsert(int index, IIOImage image, ImageWriteParam param)`

  Записывает изображение в файл с несколькими изображениями.

- `boolean canInsertImage(int index)`

  Возвращает значение `true`, если возможно включить в файл изображение с заданным индексом.

- `ImageWriterSpi getOriginatingProvider()`

  Возвращает поставщика услуг, создавшего данный объект записи.

API javax.imageio.IIOImage 1.4

- `IIOImage(RenderedImage image, List thumbnails, IIOMetadata metadata)`

  Создает объект `IIOImage` на основе заданного изображения, необязательных пиктограмм изображений и метаданных.

Управление изображениями

Предположим, что необходимо улучшить внешний вид какого-то изображения. Для этого нужно получить доступ к пикселям изображения и заменить их другими. Не исключено также, что придется создать новые пиксели "с нуля", например, для отображения результатов измерений или математических вычислений. Класс `BufferedImage` дает возможность управлять пикселями изображения, а классы, которые реализуют интерфейс `BufferedImageOp`, позволяют преобразовывать изображения.



В JDK 1.0 предлагалась совершенно другая и гораздо более сложная среда для работы с изображениями, которая была оптимизирована под *инкрементную визуализацию* (incremental rendering) изображений, загружавшихся из Web строка за строкой. Управлять такими изображениями в этой среде было, однако, очень не просто. Поэтому в настоящей книге данная среда рассматриваться не будет.

Создание растровых изображений

Большинство обрабатываемых изображений считываются из графического файла, созданного с помощью аппаратных средств (например, цифровой камеры или сканера) либо с помощью программных средств (например, графических пакетов). В этом разделе рассматриваются совершенно другие способы создания изображений, предполагающие формирование в каждый конкретный момент времени только одного пикселя.

Для создания изображения следует сформировать уже известным вам способом объект BufferedImage:

```
image = new BufferedImage(width, height, BufferedImage.TYPE_INT_ARGB);
```

Далее нужно вызвать метод getRaster() для получения объекта типа WritableRaster, который используется для доступа к пикселям изображения и их изменения:

```
WritableRaster raster = image.getRaster();
```

Метод setPixel() позволяет задавать значение для отдельного пикселя. Однако при его применении возникает сложность, состоящая в том, что пикселю нельзя непосредственно присвоить значение Color. Необходимо знать способ указания цветов для конкретного *типа* буферизованного изображения. Если изображение имеет тип TYPE_INT_ARGB, то каждый пиксель описывается четырьмя значениями (в диапазоне от 0 до 255): для красной, зеленой и синей составляющих, а также для альфа-канала. Эти значения нужно передать пикселю в виде массива из четырех элементов целого типа:

```
int[] black = {0, 0, 0, 255};
raster.setPixel(i, j, black);
```

В терминах Java 2D API эти значения называются *выборочными значениями* (sample values) пикселя.



Метод setPixel может принимать в качестве параметра массив типа float[] или массив типа double[], но учтите, что значения его элементов *не* должны быть нормализованными:

```
float[] red = {1.0F, 0.0F, 0.0F, 1.0F};
raster.setPixel(i, j, red); // Ошибка!
```

Для элементов такого массива, несмотря на его тип, все равно придется указать значения между 0 и 255.

Для установки значений прямоугольной группы пикселей можно в параметрах метода setPixels() указать начальное расположение, ширину и высоту прямоугольника, а также массив, содержащий выборочные значения цветов этой группы пикселей. Например, если изображение имеет тип TYPE_INT_ARGB, следует указать величины красной, зеленой и синей составляющих и альфа-значение для первого пикселя, потом те же данные для второго пикселя и т.д.

```
int[] pixels = new int[4 * width * height];
pixels[0] = ... // значение красного цвета для первого пикселя
pixels[1] = ... // значение зеленого цвета для первого пикселя
pixels[2] = ... // значение синего цвета для первого пикселя
pixels[3] = ... // альфа-значение для первого пикселя
...
raster.setPixels(x, y, width, height, pixels);
```

Для считывания значения пикселя используется 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 группы пикселей:

```
raster.getPixels(x, y, width, height, samples);
```

Методы getPixel()/setPixel() можно использовать и для другого типа изображения, для которого известен способ представления значений пикселей. Другими

словами, чтобы применять эти методы, нужно знать способ кодирования значений пикселей в конкретном типе изображения.

Управлять изображением становится немного сложнее, если его тип неизвестен. Каждый тип изображений имеет *цветовую модель*, в соответствии с которой можно преобразовать выборочные значения в цветовую модель RGB.



На самом деле цветовая модель RGB не является стандартом. Точное представление цветов зависит от характеристик используемого аппаратного обеспечения. Цифровые камеры, сканеры, мониторы и жидкокристаллические дисплеи имеют совершенно разные характеристики. В результате одно и то же RGB-значение выглядит по-разному в различных устройствах. Международный консорциум по цветам (International Color Consortium — ICC) (`http://www.color.org`) рекомендует прилагать ко всем цветовым данным IIC-профиль, который указывает степень соответствия цветов стандартной форме их представления на основе 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 типов освещения и цветов. 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 книги Джеймса Фоли (James D. Foley), Андриса ван Дама (Andries van Dam), Стивена Фэйнера (Steven K. Feiner) и др. *Computer Graphics: Principles and Practice, Second Edititon*.)

ICC-профили имеют очень сложную структуру. Поэтому наряду с ними используется более простой стандарт sRGB (`http://www.w3.org/Graphics/Color/sRGB.html`), который указывает точное соответствие значений цветов RGB спецификации 1931 CIE XYZ. Он разработан специально для стандартных цветных мониторов и используется в Java 2D API для преобразования цветов RGB в значения цветов других систем.

Для получения цветовой модели используется метод `getColorModel()`:

```
ColorModel model = image.getColorModel();
```

Для определения цвета пикселя нужно вызвать метод `getDataElements()` класса `Raster`, который возвращает объект с описанием цвета:

```
Object data = raster.getDataElements(x, y, null);
```



Объект, который возвращается методом `getDataElements()`, в действительности является массивом выборочных значений. Это объясняет, почему метод `getDataElements()` так называется, но для обработки данного объекта знать об этом вовсе не обязательно.

С помощью цветовой модели можно преобразовывать объекты в стандартные ARGB-величины. Метод `getRGB()` возвращает значение `int`, которое содержит значения альфа-канала, красного, зеленого и синего цветов, упакованные в четыре блока по 8 бит в каждом. С помощью конструктора `Color(int argb, boolean hasAlpha)` можно создать на основе этого целого числа объект типа `Color`:

```
int argb = model.getRGB(data);
Color color = new Color(argb, true);
```

Если для пикселя необходимо установить определенный цвет, то нужно выполнить описанные выше действия в обратном порядке. Метод `getRGB()` класса `Color` возвращает целочисленное комбинированное значение для красной, зеленой и синей составляющих и альфа-канала. Именно его нужно использовать в качестве параметра для метода `getDataElements()` класса `ColorModel`, который, в свою очередь, возвратит

объект с описанием цвета пикселя в используемой цветовой модели. Далее этот объект необходимо передать методу `setDataElements()` класса `WritableRaster`:

```
int argb = color.getRGB();
Object data = model.getDataElements(argb, null);
raster.setDataElements(x, y, data);
```

Для иллюстрации способа создания изображения путем обработки отдельных пикселей рассмотрим пример программы создания фрактального множества Мандельброта. Код программы приведен в листинге 7.10, а результат ее выполнения показан на рис. 7.28.

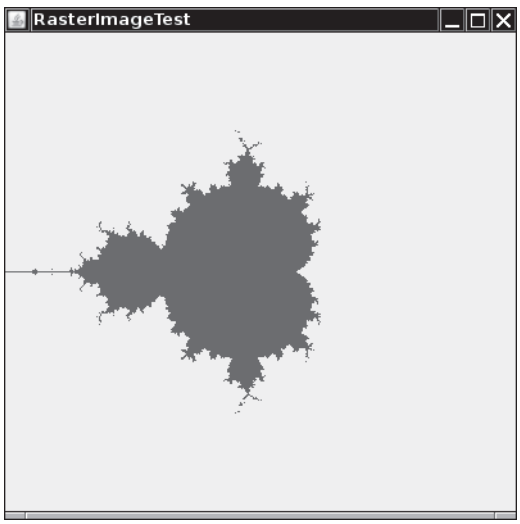


Рис. 7.28. Множество Мандельброта

Множество Мандельброта формируется путем связывания каждой точки плоскости с некоторой последовательностью цифр. Если последовательность ограничена, точка отмечается цветом. Незакрашенные точки плоскости связаны с неограниченными последовательностями цифр и остаются прозрачными.

Ниже показано, как можно создать 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:

$$x_{new} = x^2 - y^2 + a,$$
$$y_{new} = 2xy + b$$

В случае получения для x или y значения больше 2, последовательность будет превращаться в бесконечную и в цвет будут окрашиваться только те пиксели, которые соответствуют ведущим к ограниченной последовательности точкам `(a,b)`. (Формулы для вычисления числовых последовательностей берутся целиком из области математики сложных чисел. Мы просто пользуемся ими в таком, каком они и есть виде. Получить больше информации о математике фракталов можно, например, по адресу `http://classes.yale.edu/fractals/`.)

В примере программы, код которой приведен в листинге 7.6, демонстрируется использование класса `ColorModel` для преобразования значений цвета `Color` в выборочные данные пикселей. Этот процесс не зависит от типа изображения. Попробуйте поэкспериментировать и измените тип цвета для буферизированного изображения на

TYPE_BYTE_GRAY. В состав кода не нужно будет вносить никаких других изменений, так как цветовая модель изображения автоматически позаботится обо всех преобразованиях цветов в выборочные значения.

Листинг 7.6. Содержимое файла `RasterImageTest.java`

```
1.  import java.awt.*;
2.  import java.awt.image.*;
3.  import javax.swing.*;
4.  /**
5.   * Данная программа демонстрирует создание
6.   * изображения из отдельных пикселей.
7.   * @version 1.13 2007-08-16
8.   * @author Cay Horstmann
9.   */
10. public class MandelbrotTest
11. {
12.   public static void main(String[] args)
13.   {
14.     EventQueue.invokeLater(new Runnable()
15.      {
16.      public void run()
17.      {
18.        JFrame frame = new MandelbrotFrame();
19.        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20.        frame.setVisible(true);
21.      }
22.    });
23.   }
24. }
25.
26. /**
27.  * Фрейм с изображением множества Мандельброта.
28.  */
29. class RasterImageFrame extends JFrame
30. {
31.   public RasterImageFrame()
32.   {
33.     setTitle("RasterImageTest");
34.     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
35.     BufferedImage image =
36.       makeMandelbrot(DEFAULT_WIDTH, DEFAULT_HEIGHT);
37.     add(new JLabel(new ImageIcon(image)));
38.   }
39.
40.   /**
41.    * Создание изображения множества Мандельброта.
42.    * @param width Ширина
43.    * @parah height Высота
44.    * @return Изображение
45.   */
46.   public BufferedImage makeMandelbrot(int width, int height)
47.   {
48.     BufferedImage image = new BufferedImage(width, height,
49.       BufferedImage.TYPE_INT_ARGB);
50.     WritableRaster raster = image.getRaster();
51.     ColorModel model = image.getColorModel();
```

```
52.       Color fractalColor = Color.red;
53.       int argb = fractalColor.getRGB();
54.       Object colorData = model.getDataElements(argb, null);
55.
56.       for (int i = 0; i < width; i++)
57.          for (int j = 0; j < height; j++)
58.          {
59.             double a = XMIN + i * (XMAX - XMIN) / width;
60.             double b = YMIN + j * (YMAX - YMIN) / height;
61.             if (!escapesToInfinity(a, b))
62.                raster.setDataElements(i, j, colorData);
63.          }
64.       return image;
65.    }
66.
67.    private boolean escapesToInfinity(double a, double b)
68.    {
69.       double x = 0.0;
70.       double y = 0.0;
71.       int iterations = 0;
72.       while (x <= 2 && y <= 2 && iterations < MAX_ITERATIONS)
73.       {
74.         double xnew = x * x - y * y + a;
75.         double ynew = 2 * x * y + b;
76.         x = xnew;
77.         y = ynew;
78.         iterations++;
79.       }
80.       return x > 2 || y > 2;
81.    }
82.
83.    private static final double XMIN = -2;
84.    private static final double XMAX = 2;
85.    private static final double YMIN = -2;
86.    private static final double YMAX = 2;
87.    private static final int MAX_ITERATIONS = 16;
88.    private static final int DEFAULT_WIDTH = 400;
89.    private static final int DEFAULT_HEIGHT = 400;
90. }
```

API java.awt.image.BufferedImage 1.2

- `BufferedImage(int width, int height, int imageType)`
  Создает объект буферизированного изображения.

  Параметры: `width, height` Размеры изображения.
  `imageType` Тип изображения. К числу наиболее распространенных типов относятся: `TYPE_INT_RGB`, `TYPE_INT_ARGB`, `TYPE_BYTE_GRAY` и `TYPE_BYTE_INDEXED`.

- `ColorModel getColorModel()`
  Возвращает цветовую модель изображения.

- `WritableRaster getRaster()`
  Возвращает растр для доступа к пикселям изображения и их изменения.

API java.awt.image.Raster 1.2

- `Object getDataElements(int x, int y, Object data)`

  Возвращает выборочные данные для точки растра в виде массива, тип элементов и длина которого зависят от цветовой модели. Если значение параметра `data` не равно `null`, то предполагается, что это значение является массивом для хранения выборочных данных, причем массив заполнен. А если значение параметра `data` равно `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)`

  Возвращает выборочные значения для точки растра или прямоугольной области. Значения помещаются в массив, длина которого зависит от цветовой модели. Если параметр `sampleValues` равен `null`, создается новый массив. Данные методы применимы только в том случае, если разработчику известен порядок интерпретации выборочных значений для данной цветовой модели.

API java.awt.image.WritableRaster 1.2

- `void setDataElements(int x, int y, Object data)`

  Задает выборочные данные для точки растра. `data` — массив выборочных значений для пикселя. Тип элементов и длина этого массива зависит от 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)`

  Предоставляет выборочные значения для точки растра или группы точек, составляющих прямоугольную область. Данные методы применимы только тогда, когда разработчик знает принцип кодировки выборочных значений для цветовой модели.

API java.awt.image.ColorModel 1.2

- `int getRGB(Object data)`

  Возвращает ARGB-значение, которое соответствует выборочным данным, переданным в массиве `data`, тип элементов и длина которого зависит от цветовой модели.

- `Object getDataElements(int argb, Object data)`

  Возвращает выборочные данные для указанного цвета `argb`; если значение параметра `data` не равно `null`, то предполагается, что этот массив имеет подходящую длину для хранения данных и заполнен ими, а если параметр `data` равен `null`, то создается новый массив. Этот массив `data` заполняется выборочными данными для пикселя. Его тип элементов и длина массива зависит от цветовой модели.

API java.awt.Color 1.0

- `Color(int argb, boolean hasAlpha)` 1.2

  Создает цвет на основе ARGB-значения, если параметр `hasAlpha` имеет значение `true`, или на основе RGB-значения, если параметр `hasAlpha` имеет значение `false`.

- `int getRGB()`

  Возвращает ARGB-значение, соответствующее данному цвету.

Фильтрация изображений

В предыдущем разделе рассматривался процесс создания рисунка с нуля. Однако пользователю часто необходимо обработать уже имеющееся изображение.

Конечно, можно прочитать графические данные, используя методы `getPixel()`/`getDataElements()`, рассмотренные в предыдущем разделе, затем преобразовать их и записать. Но, к счастью, в Java 2D API поддерживаются *фильтры* (filters), которые автоматически выполняют многие рутинные операции обработки изображений.

Все объекты, предназначенные для обработки изображений, реализуют интерфейс `BufferedImageOp`. Для преобразования одного изображения в другое после создания объекта-операции нужно вызвать метод `filter()`.

```
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 имеют этой возможности.

Ниже приведены пять классов, которые реализуют интерфейс `BufferedImageOp`:

```
AffineTransformOp
RescaleOp
LookupOp
ColorConvertOp
ConvolveOp
```

Класс `AffineTransformOp` выполняет аффинное преобразование пикселей. Например, приведенный ниже фрагмент кода позволяет 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);
```

Конструктор `AffineTransformOp` использует в качестве параметров объект аффинного преобразования и стратегию *интерполяции* (interpolation). Интерполяция необходима для определения дискретных координат пикселей в конечном преобразованном изображении, если полученные после преобразования непрерывные значения координат исходных пикселей находятся где-то между дискретными пикселями конечного преобразованного изображения.

Например, при повороте исходных пикселей их новые непрерывные координаты не будут точно соответствовать дискретным координатам пикселей конечного преобразованного изображения. Для интерполяции используются две стратегии: билинейная стратегия `AffineTransformOp.TYPE_BILINEAR` и стратегия ближайшего соседа `AffineTransformOp.TYPE_NEAREST_NEIGHBOR`. Билинейная интерполяция выполняется дольше, но позволяет получать более качественный внешний вид.

Программа, код которой приведен в листинге 7.7, позволяет поворачивать изображение на 5° (рис. 7.29).

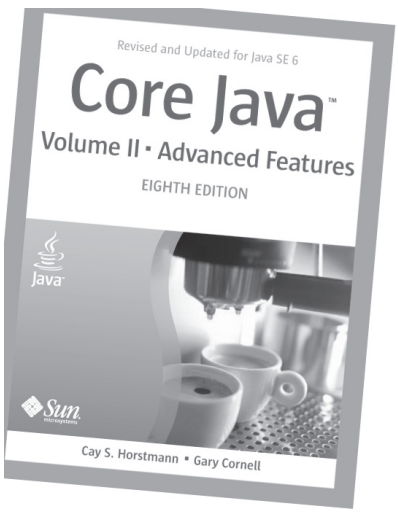


Рис. 7.29. Поворот изображения

Класс `RescaleOp` выполняет для каждого цветового компонента изображения следующую операцию по изменению масштаба:

$$x_{new} = ax + b$$

(На компоненты альфа эта операция не распространяется). Изменение масштаба с `a>1` приводит к увеличению яркости изображения. Объект `RescaleOp` создается на основе указанных параметров масштабирования и необязательных параметров рисования. В листинге 7.7 используются приведенные ниже операторы:

```
float a = 1.1f;
float 20.0f;
RescaleOp op = new RescaleOp(a, b, null);
```

Однако еще также можно и предоставлять отдельные значения масштабирования для каждого цветового компонента (как именно — можно узнать в приведенных в конце этого раздела врезках API).

Конструктор `LookupOp` позволяет создать произвольное отображение выборочных значений. Для этого используется таблица, в которой указаны правила отображения

для каждого выборочного значения. В рассматриваемом здесь примере вычисляется *негатив* заданного изображения. Для этого каждый цвет `c` заменяется цветом `255-c`.

В качестве параметров конструктор `LookupOp` использует объект-таблицу преобразования `LookupTable` и необязательную карту с параметрами рисования. Класс `LookupTable` является абстрактным и имеет два конкретных подкласса: `ByteLookupTable` и `ShortLookupTable`. Поскольку значения RGB представляются в виде байтов, возможностей класса `ByteLookupTable` должно было бы вполне хватить. Однако из обнаруженного дефекта, более подробно о котором можно узнать по адресу `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6183251`, вместо него все-таки используется класс `ShortLookupTable`. Поэтому получается, что `LookOp` конструируется в нашей демонстрационной программе следующим образом:

```
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 применяется к каждому цветовому компоненту отдельно, но не применяется к компонентам альфа-канала. Еще также можно предоставлять и отдельные таблицы преобразования для каждого цветового компонента (как именно — можно узнать в приведенных в конце этого раздела врезках API).



Класс `LookupOp` нельзя применять к изображениям с индексированной цифровой моделью, потому что в таких рисунках каждое выборочное значение задается на основе смещения в цветовой палитре.

Для преобразования цветового пространства используется класс `ColorConvertOp`, который здесь не рассматривается.

Для наиболее мощных преобразований предназначен объект `ConvolveOp`, который выполняет математическую операцию *свертки* (convolution). Мы не будем углубляться в математические подробности выполнения операции свертки, но понять основную идею несложно. Рассмотрим *фильтр размытия* (blur filter), показанный на рис. 7.30.



Рис. 7.30. Размытие изображения

Размытие достигается за счет 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* (kernel):

$$\begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}$$

Ядро свертки представляет собой матрицу с весами соседних значений. Например, приведенное выше ядро приводит к размытию изображения. Ядро, показанное ниже, выполняет операцию *выделения контуров* (edge detection), т.е. определяет участки, в которых происходит изменение цветов (рис. 7.31):

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Выделение контуров имеет очень большое значение для обработки фотографических изображений.

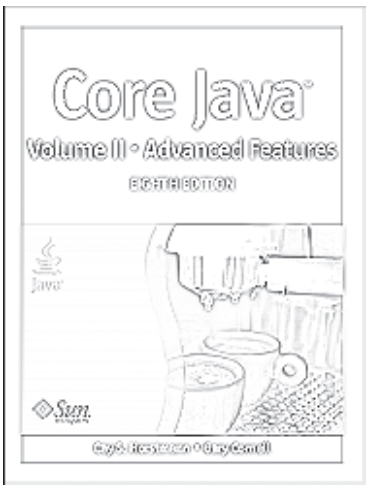


Рис. 7.31. Выделение контуров и применение инверсии

Для создания объекта свертки нужно прежде сформировать массив значений для объекта-ядра. Затем на основе этого ядра можно создать объект-операцию свертки и использовать ее для фильтрации изображения:

```
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);
```

Программа, код которой приведен в листинге 7.7, предоставляет пользователю возможность загружать файлы формата GIF и JPEG и выполнять над ними описанные выше операции. Простота кода этой программы объясняется тем, что в Java 2D API предусмотрены очень мощные средства обработки изображений.

Листинг 7.7. Содержимое файла `ImageProcessingTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.awt.geom.*;
4.  import java.awt.image.*;
5.  import java.io.*;
6.  import javax.imageio.*;
7.  import javax.swing.*;
8.  import javax.swing.filechooser.*;
9.
10. /**
11.  * Пример использования различных операций над изображениями.
12.  * @version 1.03 2007-08-16
13.  * @author Cay Horstmann
14.  */
15. public class ImageProcessingTest
16. {
17.    public static void main(String[] args)
18.    {
19.     EventQueue.invokeLater(new Runnable()
20.      {
21.       public void run()
22.       {
23.          JFrame frame = new ImageProcessingFrame();
24.          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25.          frame.setVisible(true);
26.        }
27.       });
28.    }
29. }
30. /**
31.  * Фрейм с меню для загрузки изображения и выбора для него
32.  * различных операций преобразования, а также компонентом
33.  * для отображения результирующего изображения.
34.  */
35. class ImageProcessingFrame extends JFrame
36. {
37.    public ImageProcessingFrame()
38.    {
39.       setTitle("ImageProcessingTest");
40.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
41.       add(new JComponent()
42.         {
43.            public void paintComponent(Graphics g)
44.            {
45.               if (image != null) g.drawImage(image, 0, 0, null);
46.            }
47.         });
```

```
48.       JMenu fileMenu = new JMenu("File");
49.       JMenuItem openItem = new JMenuItem("Open");
50.       openItem.addActionListener(new ActionListener()
51.          {
52.             public void actionPerformed(ActionEvent event)
53.             {
54.                openFile();
55.             }
56.          });
57.       fileMenu.add(openItem);
58.
59.       JMenuItem exitItem = new JMenuItem("Exit");
60.       exitItem.addActionListener(new  ActionListener()
61.          {
62.             public void actionPerformed(ActionEvent event)
63.             {
64.                System.exit(0);
65.             }
66.          });
67.       fileMenu.add(exitItem);
68.
69.       JMenu editMenu = new JMenu("Edit");
70.       JMenuItem blurItem = new JMenuItem("Blur");
71.       blurItem.addActionListener(new ActionListener()
72.          {
73.             public void actionPerformed(ActionEvent event)
74.             {
75.                float weight = 1.0f / 9.0f;
76.                float[] elements = new float[9];
77.                for (int i = 0; i < 9; i++)
78.                  elements[i] = weight;
79.                convolve(elements);
80.             }
81.          });
82.       editMenu.add(blurItem);
83.
84.       JMenuItem sharpenItem = new JMenuItem("Sharpen");
85.       sharpenItem.addActionListener(new ActionListener()
86.          {
87.             public void actionPerformed(ActionEvent event)
88.             {
89.                float[] elements =
90.                {
91.                   0.0f, -1.0f, 0.0f,
92.                   -1.0f,  5.f, -1.0f,
93.                   0.0f, -1.0f, 0.0f
94.                };
95.                convolve(elements);
96.             }
97.          });
98.       editMenu.add(sharpenItem);
99.
100.      JMenuItem brightenItem = new JMenuItem("Brighten");
101.      brightenItem.addActionListener(new ActionListener()
102.         {
103.            public void actionPerformed(ActionEvent event)
104.            {
105.               float a = 1.1f;
```

```
106.            float b = -20.0f;
107.            RescaleOp op = new RescaleOp(a, b, null);
108.            filter(op);
109.         }
110.       });
111.    editMenu.add(brightenItem);
112.
113.    JMenuItem edgeDetectItem = new JMenuItem("Edge detect");
114.    edgeDetectItem.addActionListener(new ActionListener()
115.       {
116.          public void actionPerformed(ActionEvent event)
117.          {
118.             float[] elements =
119.             {
120.                0.0f, -1.0f, 0.0f,
121.                -1.0f,  4.f, -1.0f,
122.                0.0f, -1.0f, 0.0f
123.             };
124.             convolve(elements);
125.          }
126.       });
127.    editMenu.add(edgeDetectItem);
128.
129.    JMenuItem negativeItem = new JMenuItem("Negative");
130.    negativeItem.addActionListener(new ActionListener()
131.       {
132.          public void actionPerformed(ActionEvent event)
133.          {
134.             {
135.             short[] negative = new short[256 * 1];
136.             for (int i = 0; i < 256; i++)
137.                negative[i] = (short) (255 - i);
138.             ShortLookupTable table = new ShortLookupTable(0, negative);
139.             LookupOp op = new LookupOp(table, null);
140.             filter(op);
141.          }
142.       });
143.    editMenu.add(negativeItem);
144.
145.    JMenuItem rotateItem = new JMenuItem("Rotate");
146.    rotateItem.addActionListener(new ActionListener()
147.       {
148.          public void actionPerformed(ActionEvent event)
149.          {
150.             if (image == null) return;
151.             AffineTransform transform =
152.                AffineTransform.getRotateInstance(
153.                Math.toRadians(5), image.getWidth() / 2,
154.                image.getHeight() / 2);
155.             AffineTransformOp op = new AffineTransformOp(transform,
156.             AffineTransformOp.TYPE_BICUBIC);
157.             filter(op);
158.          }
159.       });
160.    editMenu.add(rotateItem);
161.
162.    JMenuBar menuBar = new JMenuBar();
```

```
163.      menuBar.add(fileMenu);
164.      menuBar.add(editMenu);
165.      setJMenuBar(menuBar);
166.    }
167.
168.    /**
169.     * Открытие файла и загрузка изображения.
170.     */
171.    public void openFile()
172.    {
173.      JFileChooser chooser = new JFileChooser();
174.      chooser.setCurrentDirectory(new File("."));
175.      String[] extensions = ImageIO.getReaderFileSuffixes();
176.      chooser.setFileFilter(new FileNameExtensionFilter("Image files", extensions));
177.      int r = chooser.showOpenDialog(this);
178.      if (r != JFileChooser.APPROVE_OPTION) return;
179.
180.      try
181.      {
182.        Image img = ImageIO.read(chooser.getSelectedFile());
183.        image = new BufferedImage(img.getWidth(null), img.getHeight(null),
184.          BufferedImage.TYPE_INT_RGB);
185.       image.getGraphics().drawImage(img, 0, 0, null);
186.      }
187.     catch (IOException e)
188.     {
189.        JOptionPane.showMessageDialog(this, e);
190.      }
191.     repaint();
192.   }
193.
194.    /**
195.     * Применение фильтра и перерисовка изображения.
196.     * @param op Операция, которую нужно применить к изображению
197.     */
198.    private void filter(BufferedImageOp op)
199.    {
200.      if (image == null) return;
201.      image = op.filter(image, null);
202.      repaint();
203.    }
204.
205.    /**
206.     * Применение свертки и перерисовка изображения.
207.     * @param elements Ядро свертки (матрица из 9 элементов)
208.     */
209.    private void convolve(float[] elements)
210.    {
211.      Kernel kernel = new Kernel(3, 3, elements);
212.      ConvolveOp op = new ConvolveOp(kernel);
213.      filter(op);
214.    }
215.
216.    private BufferedImage image;
217.    private static final int DEFAULT_WIDTH = 400;
218.    private static final int DEFAULT_HEIGHT = 400;
219. }
```

API 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 для параметра `dest` задано значение `null`, то создается новое целевое изображение.

API java.awt.image.AffineTransformOp 1.2

- `AffineTransformOp(AffineTransform t, int interpolationType)`

  Создает объект-операцию аффинного преобразования. Для типа интерполяции может указываться одно из следующих значений: `TYPE_BILINEAR`, `TYPE_BICUBIC` или `TYPE_NEAREST_NEIGHBOR`.

API java.awt.image.RescaleOp 1.2

- `RescaleOp(float a, float b, RenderingHints hints)`
- `RescaleOp(float[] as, float[] bs, RenderingHints hints)`

  Создает оператор масштабирования, который выполняет следующую операцию по изменению масштаба: x_{new} = `a*x + b`. В случае использования первого конструктора все цветовые компоненты (кроме компонента альфа) масштабируются с одинаковыми коэффициентами. В случае применения второго оператора значения предоставляются либо для каждого цветового компонента отдельно без привлечения компонента альфа, либо как для цветовых компонентов, так и для компонента альфа.

API java.awt.image.LookupOp 1.2

- `LookupOp(LookupTable table, RenderingHints hints)`

  Создает объект-операцию подстановки для указанной таблицы преобразования.

API java.awt.image.ByteLookupTable 1.2

- `ByteLookupTable(int offset, byte[] data)`
- `ByteLookupTable(int offset, byte[][] data)`

  Создает таблицу преобразования для преобразования значений типа `byte`. Значение смещения вычитается из входных данных до выполнения преобразования. Значения в первом конструкторе применяются ко всем цветовым компонентам, но не компоненту альфа. В случае использования второго конструктора значения предоставляются либо для каждого цветового компонента отдельно без привлечения компонента альфа, либо как для цветовых компонентов, так и для компонента альфа.

API java.awt.image.ShortLookupTable 1.2

- `ShortLookupTable(int offset, short[] data)`
- `ShortLookupTable(int offset, short[][] data)`

  Создает таблицу преобразования для преобразования значений `short`. Значение смещения вычитается из входных данных до выполнения преобразования.

Значения в первом конструкторе применяются ко всем цветовым компонентам, но не компоненту альфа. В случае использования второго конструктора значения предоставляются либо для каждого цветового компонента отдельно без привлечения компонента альфа, либо как для цветовых компонентов, так и для компонента альфа.

API java.awt.image.ConvolveOp 1.2

- `ConvolveOp(Kernel kernel)`
- `ConvolveOp(Kernel kernel, int edgeCondition, RenderingHints hints)`

Создает объект-операцию свертки. Для параметра `edgeCondition` может указываться либо значение `EDGE_NO_OP`, либо значение `EDGE_ZERO_FILL`. К этим значениям нужно относиться особенно внимательно, поскольку у них нет подходящих соседних значений для вычисления свертки. По умолчанию устанавливается `EDGE_ZERO_FILL`.

API java.awt.image.Kernel 1.2

- `Kernel(int width, int height, float[] matrixElements)`

Создает ядро для указанной матрицы.

Печать

В исходной версии JDK не поддерживался вывод данных на печать. Например, нельзя было печатать из аплетов, а для печати из приложений необходимо было использовать дополнительные библиотеки независимых разработчиков. Впервые средства организации печати появились в JDK 1.1, но с их помощью можно было выводить очень простые документы довольно низкого качества. Разработчикам Web-браузеров была предоставлена специальная модель печати, позволяющая выводить окно аплета в таком виде, в каком оно отображалось на Web-странице (чем, однако, они не воспользовались).

В Java SE 1.2 были заложены основы модели печати, полностью интегрированной со средствами двухмерной графики. В Java SE 1.4 эта модель была существенно усовершенствована возможностями управления параметрами принтеров и организации заданий потоковой печати на стороне сервера.

В этом разделе рассматриваются способы печати рисунков на одной странице бумаги, способы управления многостраничной печатью, а также демонстрируются возможности модели представления изображений Java 2D API и средства создания окна предварительного просмотра.



На платформе Java поддерживается печать элементов пользовательского интерфейса, но здесь данная тема не рассматривается, поскольку это в основном интересно создателям Web-браузеров, а также программ захвата содержимого экрана, и разработчикам других подобных продуктов. Более подробную информацию по этой теме можно получить по адресу `http://java.sun.com/developer/onlineTraining/Programming/JDCBook/render.html`.

Вывод на печать графических изображений

В данном разделе рассматривается одна из наиболее распространенных задач, а именно — печать двухмерной графики. Конечно, изображение может содержать

фрагменты текста, написанного с использованием различных шрифтов, или даже полностью состоять из текста.

Для организации печати необходимо выполнить следующее:

- предоставить объект, который реализует интерфейс `Printable`;
- запустить процесс печати.

В интерфейсе `Printable` объявлен единственный метод `print()`:

```
int print(Graphics g, PageFormat format, int page)
```

Этот метод вызывается всякий раз, когда механизм печати форматирует страницу. Созданный программистом код рисует текст и изображение, которые должны быть напечатаны в данном графическом контексте `g`. Формат страницы `format` указывает размер бумаги и границы печати, а номер `page` используется для выбора отображаемой страницы.

Для запуска процесса печати применяется класс `PrinterJob`. Сначала для получения объекта-задания печати вызывается статический метод `getPrinterJob()`, а затем с помощью метода `setPrintable()` указывается объект типа `Printable`, подлежащий выводу:

```
Printable canvas = ...;
PrinterJob job = PrinterJob.getPrinterJob();
job.setPrintable(canvas);
```



Не путайте класс `PrinterJob` с устаревшим классом `PrintJob`, который управлял процессом печати в JDK 1.1.

Перед запуском задания печати нужно вызвать метод `printDialog()` для отображения диалогового окна, показанного на рис. 7.32. Это окно предоставляет пользователю возможность выбирать используемый принтер (если доступно несколько принтеров), указывать диапазон страниц для печати и выполнять другие установки.

Все параметры принтера обычно собираются в экземпляре такого класса, который реализует интерфейс `PrintRequestAttributeSet`, как, например, класс `HashPrintRequestAttributeSet`:

```
HaspPrintRequestAttributeSet attributes
   = new HaspPrintRequestAttributeSet();
...
```

Объект `attributes` необходимо передать методу `printDialog()`.

Метод `printDialog` возвращает либо значение `true`, если пользователь подтверждает выбор параметров, либо значение `false`, если пользователь отменяет внесенные изменения. В первом случае вызывается метод `print()` класса `PrinterJob` для запуска процесса печати. Он может генерировать исключение `PrinterException`. Поэтому фрагмент кода, организующий вывод на печать, выглядит так, как показано ниже.

```
if (job.printDialog(attributes))
{
   try
   {
      job.print(attributes);
   }
   catch (PrinterException exception)
   {
      . . .
   }
}
```



До появления JDK 1.4 в механизме печати Java использовались диалоговые окна параметров печати, ориентированные именно на ту платформу, на которой работал пользователь. Для отображения платформенно-ориентированного диалогового окна печати нужно вызвать метод `printDialog()` без указания параметров. (В этом случае не существует никакого способа объединения пользовательских установок в наборе атрибутов.)

Во время печати метод `print()` класса `PrinterJob` повторно вызывает метод `print()` объекта `Printable`, который связан с данной задачей печати.

Поскольку задаче печати неизвестно количество выводимых страниц, метод `print()` повторно вызывается, если он возвращает значение `Printable.PAGE_EXISTS`. Печать прекратится лишь тогда, когда метод `print()` возвратит значение `Printable.NO_SUCH_PAGE`.



Номера страниц, которые передаются методу `print`, отсчитываются, начиная с нуля.

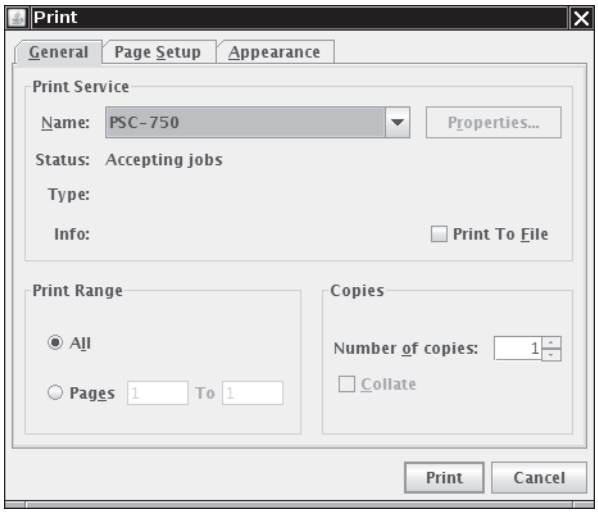


Рис. 7.32. Диалоговое окно для выбора параметров печати

Задача печати не может определить количество страниц до тех пор, пока печать не будет завершена. Поэтому при такой организации печати диалоговое окно выбора параметров не может отобразить фактический диапазон номеров страниц и выводит сообщение `Pages 1 to 1`. В следующем разделе показан способ устранения этого недостатка с помощью объекта `Book`.

Задание печати может повторно вызывать метод `print()` объекта `Printable` для *одной и той же страницы*. Поэтому не следует считать страницы внутри метода `print`, можно полагаться на номер страницы. Возможность многократного обращения к одной и той же странице предназначена для обслуживания некоторых принтеров, особенно матричных и струйных, которые используют *полосовой* (banding) принцип печати. Они печатают одну полосу за один раз, продвигают бумагу и затем печатают следующую полосу. Полосовой принцип печати может использоваться даже для лазерных принтеров, которые обычно печатают сразу всю страницу. Это дает возможность управлять размером файла спулера.

Если заданию печати требуется объект Printable для вывода полосы, следует указать область отсечения графического контекста и вызвать метод print. Таким образом, все операции рисования будут выполняться только в этой полосе, и только этот элемент изображения будет воспроизведен. Метод print() не обязательно должен знать об этих действиях при условии, что он не конфликтует с областью отсечения.



Возвращаемый методом print() объект Graphics отсекается также по границам страницы. Но рисовать за пределами границ страницы можно, если переместить область отсечения. Однако в графическом контексте принтера нужно строго соблюдать заданную область отсечения. Для дальнейшего ограничения области отсечения следует вызывать метод clip (не setClip). А если нужно временно удалить область отсечения, то в начале метода print() потребуется вызвать метод getClip(), а затем восстановить область отсечения.

Параметр PageFormat метода print() содержит информацию о печатаемой странице. Методы getWidth() и getHeight() возвращают размер бумаги, измеряемый в *пунктах* (points). Один пункт равен 1/72 дюйма, а дюйм равен 25,4 миллиметра.

Например, страница формата A4 имеет размер примерно 595 пунктов в ширину и 842 пункта в длину, а страница формата Letter — примерно 612 пунктов в ширину и 792 пункта в длину.

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 в конце раздела. Программа печатает две строки текста, которые находятся на расстоянии 72 пункта друг от друга. Запустите программу, измерьте расстояние между напечатанными строками и убедитесь в том, что оно равно точно 1 дюйму, или 25,4 мм.

Методы getWidth() и getHeight() класса PageFormat возвращают полную ширину и высоту страницы. Но печатать можно не на всей поверхности страницы. Обычно пользователи самостоятельно задают размер поля, но даже если они этого не сделают явно, то все равно поля будут определены. Дело в том, что принтеру нужно как-то удерживать листы бумаги, на которых выводится информация, и поэтому по краям остаются маленькие недоступные для печати области.

Методы getImageableWidth() и getImageableHeight() возвращают ширину и высоту доступной для печати области. Однако поля не обязательно должны быть симметричны, поэтому нужно также знать координаты верхнего левого угла доступной для печати области, которая условно показана на рис. 7.33. Эти координаты можно получить с помощью методов getImageableX() и getImageableY().



Графический контекст, который передается методу print(), обрезается по полям страницы, но начало системы координат, несмотря ни на что, остается в верхнем левом углу страницы. Поэтому имеет смысл перенести начало координат в левый верхний угол доступной для печати области. Для этого необходимо поместить в начало метода print() следующий оператор:

```
g.translate(pageFormat.getImageableX(), pageFormat.getImageableY());
```

Если вы хотите предоставить пользователю возможность определять границы страницы и задавать книжную или альбомную ориентацию, не устанавливая при этом других атрибутов печати, можете сделать это, вызвав метод pageDialog() класса PrinterJob:

```
PageFormat format = job.pageDialog(attributes);
```

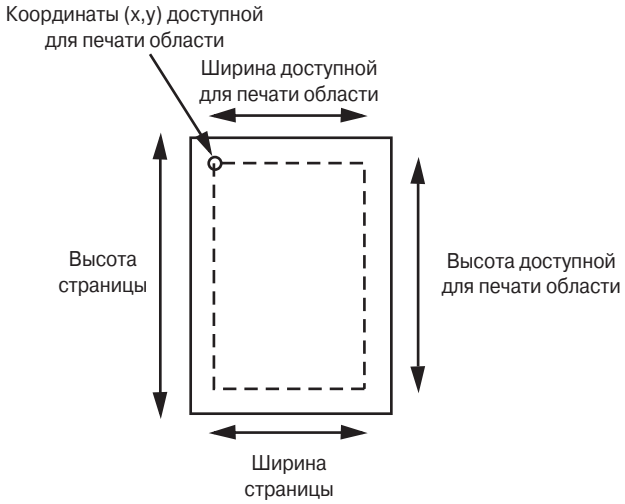


Рис. 7.33. Размеры страницы и доступной для печати области



Одна из вкладок диалогового окна параметров содержит параметры страницы (рис. 7.34), которые пользователь может просмотреть и отредактировать непосредственно перед печатью. Это особенно полезно, если в программе реализован принцип WYSIWIG, т.е. на печать направляются данные, отображаемые на экране. Метод `pageDialog()` возвращает объект `PageFormat` со значениями параметров печати, которые заданы пользователем.

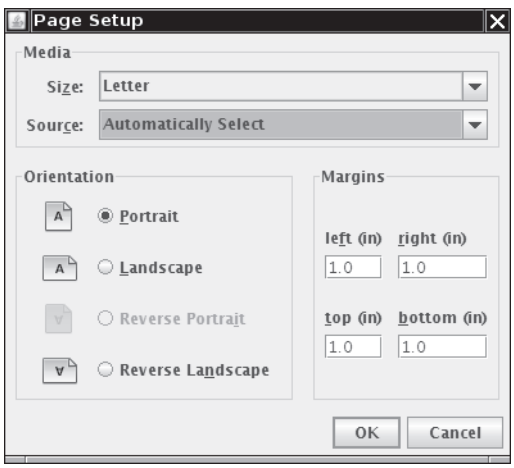


Рис. 7.34. Диалоговое окно параметров печати

В примере программы, код которой приведен в листинге 7.8, демонстрируются способы отображения набора фигур на экране и вывода их на печать. Класс `PrintPanel` расширяет класс `JPanel` и реализует интерфейс `Printable`, а его методы `paintComponent()` и `print()` вызывают один и тот же метод для фактического рисования изображения.

```
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)
   {
      // Код, предназначенный для фактического
      // рисования изображения.
      ...
   }
...
}
```

В рассматриваемом здесь примере для демонстрации вывода данных на печать используется то же изображение, которое было показано на рис. 7.20. В нем контуры символов, составляющих строку `Hello, World`, используются как область отсечения для набора линий (см. рис. 7.22).

Для запуска процесса печати щелкните на кнопке Print (Печать), а для вызова диалогового окна параметров печати — на кнопке Page setup (Параметры страницы). Полный текст программы приведен в листинге 7.8.



Для того чтобы отобразить платформенно-ориентированное диалоговое окно параметров страницы, нужно передать объект `PageFormat` методу `pageDialog()`. Этот метод создает копию объекта, изменяет ее в соответствии с пользовательскими установками и возвращает вызывающему методу.

```
PageFormat defaultFormat = printJob.defaultPage();
pageFormat selectedFormat = printJob.pageDialog(defaultFormat);
```

Листинг 7.8. Содержимое файла `PrintTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.awt.font.*;
4.  import java.awt.geom.*;
5.  import java.awt.print.*;
6.  import java.util.*;
7.  import javax.print.*;
8.  import javax.print.attribute.*;
9.  import javax.swing.*;
10.
```

```
11. /**
12.  * Пример печати двухмерной графики.
13.  * @version 1.12 2007-08-16
14.  * @author Cay Horstmann
15.  */
16. public class PrintTest
17. {
18.    public static void main(String[] args)
19.    {
20.      EventQueue.invokeLater(new Runnable()
21.       {
22.         public void run()
23.       {
24.         JFrame frame = new PrintTestFrame();
25.         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26.         frame.setVisible(true);
27.       }
28.     });
29.  }
30. }
31.
32. /**
33.  * Фрейм, содержащий панель для вывода двухмерного изображения,
34.  * а также кнопки печати и настройки параметров страницы.
35.  */
36. class PrintTestFrame extends JFrame
37. {
38.    public PrintTestFrame()
39.    {
40.       setTitle("PrintTest");
41.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
42.
43.       canvas = new PrintComponent();
44.       add(canvas, BorderLayout.CENTER);
45.
46.       attributes = new HashPrintRequestAttributeSet();
47.
48.       JPanel buttonPanel = new JPanel();
49.       JButton printButton = new JButton("Print");
50.       buttonPanel.add(printButton);
51.       printButton.addActionListener(new ActionListener()
52.          {
53.             public void actionPerformed(ActionEvent event)
54.             {
55.                try
56.                {
57.                   PrinterJob job = PrinterJob.getPrinterJob();
58.                   job.setPrintable(canvas);
59.                   if (job.printDialog(attributes))job.print(attributes);
60.                }
61.                catch (PrinterException e)
62.                {
63.                   JOptionPane.showMessageDialog(PrintTestFrame.this, e);
64.                }
65.             }
66.          });
67.
```

```
68.       JButton pageSetupButton = new JButton("Page setup");
69.       buttonPanel.add(pageSetupButton);
70.       pageSetupButton.addActionListener(new ActionListener()
71.          {
72.             public void actionPerformed(ActionEvent event)
73.             {
74.                PrinterJob job = PrinterJob.getPrinterJob();
75.                job.pageDialog(attributes);
76.             }
77.          });
78.
79.       add(buttonPanel, BorderLayout.NORTH);
80.    }
81.
82.    private PrintPanel canvas;
83.    private PrintRequestAttributeSet attributes;
84.
85.    private static final int DEFAULT_WIDTH = 300;
86.    private static final int DEFAULT_HEIGHT = 300;
87. }
88.
89. /**
90.  * Компонент с двухмерным изображением для
91.  * демонстрации его внешнего вида на экране
92.  * и вывода на печать.
93.  */
94. class PrintComponent extends JComponent implements Printable
95. {
96.    public void paintComponent(Graphics g)
97.    {
98.       Graphics2D g2 = (Graphics2D) g;
99.       drawPage(g2);
100.   }
101.
102.   public int print(Graphics g, PageFormat pf, int page)
103.      throws PrinterException
104.   {
105.      if (page >= 1) return Printable.NO_SUCH_PAGE;
106.      Graphics2D g2 = (Graphics2D) g;
107.      g2.translate(pf.getImageableX(), pf.getImageableY());
108.      g2.draw(new Rectangle2D.Double(0, 0, pf.getImageableWidth(),
109.         pf.getImageableHeight()));
110.
111.      drawPage(g2);
112.      return Printable.PAGE_EXISTS;
113.   }
114.
115.   /**
116.    * Код рисования страницы на экране
117.    * и в графическом контексте печати.
118.    * @param g2 Графический контекст
119.    */
120.   public void drawPage(Graphics2D g2)
121.   {
122.      FontRenderContext context = g2.getFontRenderContext();
123.      Font f = new Font("Serif", Font.PLAIN, 72);
124.      GeneralPath clipShape = new GeneralPath();
```

```
125.      TextLayout layout = new TextLayout("Hello", f, context);
126.      AffineTransform transform =
127.        AffineTransform.getTranslateInstance(0, 72);
128.      Shape outline = layout.getOutline(transform);
129.      clipShape.append(outline, false);
130.
131.      layout = new TextLayout("World", f, context);
132.      transform  = AffineTransform.getTranslateInstance(0, 144);
133.      outline = layout.getOutline(transform);
134.      clipShape.append(outline, false);
135.
136.      g2.draw(clipShape);
137.      g2.clip(clipShape);
138.
139.      final int NLINES =50;
140.      Point2D p = new Point2D.Double(0, 0);
141.      for (int i = 0; i < NLINES; i++)
142.      {
143.        double x = (2 * getWidth() * i) / NLINES;
144.        double y = (2 * getHeight() * (NLINES - 1 - i))
145.          / NLINES;
146.        Point2D q = new Point2D.Double(x, y);
147.        g2.draw(new Line2D.Double(p, q));
148.      }
149.    }
150. }
```

API java.awt.print.Printable 1.2

- `int print(Graphics g, PageFormat format, int pageNumber)`

  Отображает страницу и возвращает значение `PAGE_EXISTS` или `NO_SUCH_PAGE`.

Параметры:	`g`	Графический контекст отображаемой страницы
	`format`	Формат страницы
	`pageNumber`	Номер требуемой страницы

API java.awt.print.PrinterJob 1.2

- `static PrinterJob getPrinterJob()`

  Возвращает объект-задание печати.

- `PageFormat defaultPage()`

  Возвращает установленный по умолчанию формат страницы для данного принтера.

- `boolean printDialog(PrintRequestAttributeSet attributes)`
- `boolean printDialog()`

  Открывает диалоговое окно, позволяющее пользователю выбирать страницы для печати и выполнять настройку принтера. Первый из указанных методов отображает платформенно-независимое диалоговое окно, а второй — платформенно-ориентированное окно. Оба метода изменяют объект `attributes` с учетом пользовательских настроек и возвращают значение `true`, если пользователь подтвердил установленные параметры печати.

- `PageFormat pageDialog(PrintRequestAttributeSet attributes)`
- `PageFormat pageDialog(PageFormat defaults)`

  Выводит диалоговое окно для установки параметров страницы. Первый из указанных методов отображает платформенно-независимое диалоговое окно, а второй — платформенно-ориентированное окно. Оба метода возвращают объект `PageFormat`, определяющий формат, указанный пользователем в диалоговом окне. Первый метод модифицирует объект `attributes` так, чтобы тот отражал установки пользователя. Второй метод не модифицирует объект `defaults`.

- `void setPrintable(Printable p)`
- `void setPrintable(Printable p, PageFormat format)`

  Задает для данного задания печати объект `Printable` и необязательный формат страницы.

- `void print()`
- `void print(PrintRequestAttributeSet attributes)`

  Выводит на печать текущий объект `Printable`, вызывая его метод `print()` и передавая отображаемые страницы принтеру. Данный процесс продолжается до тех пор, пока страницы не будут исчерпаны.

API **java.awt.print.PageFormat 1.2**

- `double getWidth()`
- `double getHeight()`

  Возвращает ширину и высоту страницы.

- `double getImageWidth()`
- `double getImageHeight()`

  Возвращает ширину и высоту доступной для печати области страницы.

- `double getImageableX()`
- `double getImageableY()`

  Возвращает координаты верхнего левого угла доступной для печати области страницы.

- `int getOrientation()`

  Возвращает одно из следующих значений, определяющих ориентацию страницы: `PORTRAIT`, `LANDSCAPE` или `REVERSE_LANDSCAPE`. Ориентация "прозрачна" для программиста, так как формат страниц установки графического контекста автоматически учитывает ориентацию.

Многостраничная печать

На практике нежелательно передавать объекты типа `Printable` заданию печати без предварительной обработки. Сначала следует получить экземпляр класса, который реализует интерфейс `Pageable`. Платформа Java для этого предусматривает класс-книгу `Book`, который состоит из классов-разделов типа `Printable`. Итак, создадим объект-книгу, указывая объекты-разделы и нумерацию страниц так, как показано ниже:

```
Book book = new Book();
Printable coverPage = ...;
Printable bodyPages = ...;
book.append(coverPage, pageFormat); // Добавление 1-й страницы.
book.append(coverPage, pageFormat, pageCount);
```

Затем следует использовать метод `setPageable()` для передачи заданию печати объекта-книги `Book`:

```
printJob.setPageable(book);
```

Теперь заданию печати точно известно количество печатаемых страниц и в диалоговом окне печати точно отображаются их номера, которыми пользователь может самостоятельно управлять, выбирая весь диапазон или только его часть.



Когда задание печати вызывает метод `print()` объектов-разделов типа `Printable`, при этом передается текущий номер страницы, отсчитываемый в пределах *книги*, а не текущего *раздела*. Это очень неудобно, поскольку в каждом разделе должна храниться информация о количестве страниц в предыдущих разделах.

С точки зрения программиста, наибольшую сложность при использовании класса книги `Book` представляет подсчет и сохранение информации о количестве страниц в каждом разделе в ходе процесса печати. В объекте типа `Printable` нужно создать *алгоритм обработки макета* печатаемой страницы. Обработку макета нужно выполнить перед началом печати для определения задания печати и точек разрыва. Информацию о макете страницы можно сохранить для использования во время печати.

Необходимо учитывать, что пользователь может изменить формат страницы. В таком случае придется повторно рассчитать макет страницы, даже если печатаемая информация не изменилась.

В примере программы, код которой приведен в листинге 7.9, демонстрируется способ печати нескольких страниц. Эта программа выводит плакат, состоящий из строки символов очень большого размера, которые располагаются на нескольких страницах (рис. 7.35). После печати нескольких страниц поля отрезаются, а страницы склеиваются вместе.

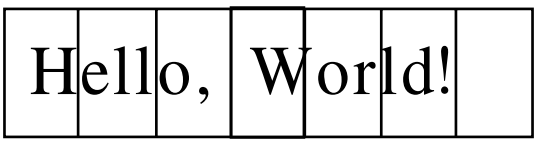


Рис. 7.35. Плакат

Метод `layoutPages()` класса `Banner` рассчитывает макет страницы, а затем макет строки сообщения с размером шрифта 72 пункта. Далее вычисляется высота результирующей строки и сравнивается с высотой печатаемой области страницы. На основании этих двух измерений определяется коэффициент масштабирования. При выводе на печать строка будет увеличена в соответствии с этим коэффициентом.



Для точного размещения на странице печатаемой информации требуется доступ к графическому контексту принтера. К сожалению, не предусмотрено никакого способа получения графического контекста до тех пор, пока действительно не начнется процесс печати. В программе, рассматриваемой здесь в качестве примера, все действия выполняются с экранным графическим контекстом в надежде, что характеристики шрифтов на экране и на печатной странице совпадают.

Метод `getPageCount()` класса `Banner` сначала вызывает метод определения макета страницы. Затем он масштабирует ширину строки и делит ее на ширину доступной

для печати области отдельной страницы. Частное от такого деления, округленное до следующего целого значения, определяет искомое количество страниц.

Исходя из сказанного выше, может показаться, что напечатать плакат довольно сложно, поскольку отдельные части символа могут располагаться на разных страницах. Однако благодаря мощным средствам Java 2D API, эта задача решается очень просто. Для получения определенной страницы используется метод `translate()` класса `Graphics2D`, который смещает верхний левый угол строки на заданное расстояние влево. Затем устанавливается прямоугольная область отсечения с размерами текущей страницы (рис. 7.36), после чего графический контекст масштабируется в соответствии с масштабным коэффициентом, вычисленным с помощью метода компоновки страницы.



Рис. 7.36. Алгоритм постраничной печати большого плаката

Этот пример иллюстрирует возможности функций преобразования, благодаря которым код рисования остается очень 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 применение преобразований для организации предварительного просмотра печатаемой страницы.

Предварительный просмотр

В большинстве коммерческих программ предусмотрен механизм предварительного просмотра, который позволяет оценить внешний вид страницы на экране компьютера до запуска процесса печати. Это позволяет не тратить бумагу на печать страниц с неудовлетворительным расположением элементов. В классах печати платформы Java не предусмотрено стандартного диалогового окна предварительного просмотра. Однако окно, подобное показанному на рис. 7.37, можно довольно легко создать самостоятельно. В этом разделе рассматривается один из способов создания такого окна. Приведенный в листинге 7.9 класс `PrintPreviewDialog` можно рассматривать как универсальный; он обеспечивает предварительный просмотр внешнего вида любых типов печатаемых страниц.

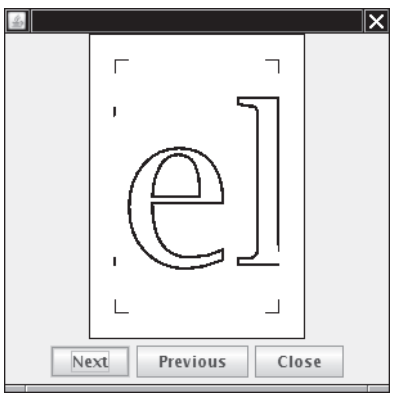


Рис. 7.37. Диалоговое окно предварительного просмотра

Для создания класса PrintPreviewDialog используется либо класс Printable, либо класс Book, а также объект PageFormat. Для отображения внешнего вида диалогового окна применяется класс PrintPreviewCanvas. При активизации кнопок Next (Следующая) и Previous (Предыдущая), предназначенных для перехода к следующей и предыдущей страницам, метод paintComponent() вызывает метод print() объекта Printable для запрашиваемой страницы.

Обычно метод print() отображает содержимое страницы в графическом контексте принтера. Однако в данном случае используется экранный графический контекст, масштабированный таким образом, чтобы печатаемая страница целиком поместилась внутри небольшой прямоугольной области экрана.

```
float xoff = ...;  // левая сторона страницы
float yoff = ...;  // верхняя сторона страницы
float scale = ...; // масштабный коэффициент для подгонки
                   // размеров страницы к размерам
                   // прямоугольной области экрана
g2.translate(xoff, yoff);
g2.scale(sclae, scale);
Printable printable = book.getPrintable(currentPage);
printable.print(g2, pageFormat, currentPage);
```

В данном случае метод print() работает так же, как и при фактической печати страницы, т.е. он просто рисует страницу в графическом контексте, которая в уменьшенном виде отображается на экране. Этот пример очень хорошо демонстрирует возможности графической модели Java 2D.

В листинге 7.9 содержится код программы печати большого плаката с диалоговым окном предварительного просмотра. Введите строку Hello, World! в текстовом поле, оцените внешний вид печатаемой страницы, а затем напечатайте плакат.

Листинг 7.9. Содержимое файла BookTest.java

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.awt.font.*;
4.  import java.awt.geom.*;
5.  import java.awt.print.*;
6.  import javax.print.attribute.*;
7.  import javax.swing.*;
8.
9.  /**
10.  * Пример программы, которая демонстрирует печать
11.  * многостраничного объекта-книги. В качестве книги
12.  * используется плакат со строкой большого размера,
13.  * части которой располагаются на разных страницах.
14.  * Эта программа также содержит диалоговое окно
15.  * предварительного просмотра
16.  * @version 1.12 2007-08-16
17.  * @author Cay Horstmann
18.  */
19. public class BookTest
20. {
21.    public static void main(String[] args)
22.    {
23.      EventQueue.invokeLater(new Runnable()
24.       {
```

```
25.         public void run()
26.         {
27.           JFrame frame = new BookTestFrame();
28.           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29.           frame.setVisible(true);
30.         }
31.       });
32.    }
33. }
34. /**
35.  * Фрейм с полем редактирования для ввода текста плаката
36.  * и кнопками печати, установки параметров страницы
37.  * и предварительного просмотра.
38.  */
39. class BookTestFrame extends JFrame
40. {
41.    public BookTestFrame()
42.    {
43.       setTitle("BookTest");
44.       text = new JTextField();
45.       add(text, BorderLayout.NORTH);
46.
47.       attributes = new HashPrintRequestAttributeSet();
48.
49.       JPanel buttonPanel = new JPanel();
50.       JButton printButton = new JButton("Print");
51.       buttonPanel.add(printButton);
52.       printButton.addActionListener(new ActionListener()
53.          {
54.             public void actionPerformed(ActionEvent event)
55.             {
56.                try
57.                {
58.                   PrinterJob job = PrinterJob.getPrinterJob();
59.                   job.setPageable(makeBook());
60.                   if (job.printDialog(attributes))
61.                   {
62.                      job.print(attributes);
63.                   }
64.                }
65.                catch (PrinterException e)
66.                {
67.                   JOptionPane.showMessageDialog(
68.                      BookTestFrame.this, e);
69.                }
70.             }
71.          });
72.       JButton pageSetupButton = new JButton("Page setup");
73.       buttonPanel.add(pageSetupButton);
74.       pageSetupButton.addActionListener(new ActionListener()
75.          {
76.             public void actionPerformed(ActionEvent event)
77.             {
78.                PrinterJob job = PrinterJob.getPrinterJob();
79.                pageFormat = job.pageDialog(attributes);
80.             }
81.          });
```

```
82.       JButton printPreviewButton = new JButton("Print preview");
83.       buttonPanel.add(printPreviewButton);
84.       printPreviewButton.addActionListener(new ActionListener()
85.          {
86.             public void actionPerformed(ActionEvent event)
87.             {
88.                PrintPreviewDialog dialog =
89.                   new PrintPreviewDialog(makeBook());
90.                dialog.setVisible(true);
91.             }
92.          });
93.
94.       add(buttonPanel, BorderLayout.SOUTH);
95.       pack();
96.    }
97.
98.    /**
99.     * Создание объекта-книги с титульной
100.    * страницей и обычными страницами.
101.    */
102.   public Book makeBook()
103.   {
104.      if (pageFormat == null)
105.      {
106.         PrinterJob job = PrinterJob.getPrinterJob();
107.         pageFormat = job.defaultPage();
108.      }
109.      Book book = new Book();
110.      String message = text.getText();
111.      Banner banner = new Banner(message);
112.      int pageCount = banner.getPageCount(
113.         (Graphics2D)getGraphics(), pageFormat);
114.      book.append(new CoverPage(message + " (" +
115.         pageCount + " pages)"), pageFormat);
116.      book.append(banner, pageFormat, pageCount);
117.      return book;
118.   }
119.
120.   private JTextField text;
121.   private PageFormat pageFormat;
122.   private PrintRequestAttributeSet attributes;
123. }
124.
125. /**
126.  * Плакат, представляющий собой строку текста,
127.  * расположенную на нескольких страницах.
128.  */
129. class Banner implements Printable
130. {
131.    /**
132.     * Создание плаката.
133.     * @param m Строка
134.     */
135.    public Banner(String m)
136.    {
137.       message = m;
138.    }
```

```
139.   /**
140.    * Определение количества страниц в данном разделе.
141.    * @param g2 Графический контекст
142.    * @param pf Формат страницы
143.    * @return Требуемое количество страниц
144.    */
145.   public int getPageCount(Graphics2D g2, PageFormat pf)
146.   {
147.      if (message.equals("")) return 0;
148.      FontRenderContext context = g2.getFontRenderContext();
149.      Font f = new Font("Serif", Font.PLAIN, 72);
150.      Rectangle2D bounds = f.getStringBounds(message, context);
151.      scale = pf.getImageableHeight() / bounds.getHeight();
152.      double width = scale * bounds.getWidth();
153.      int pages = (int)Math.ceil(width / pf.getImageableWidth());
154.      return pages;
155.   }
156.
157.   public int print(Graphics g, PageFormat pf, int page)
158.      throws PrinterException
159.   {
160.      Graphics2D g2 = (Graphics2D)g;
161.      if (page > getPageCount(g2, pf))
162.         return Printable.NO_SUCH_PAGE;
163.      g2.translate(pf.getImageableX(), pf.getImageableY());
164.
165.      drawPage(g2, pf, page);
166.      return Printable.PAGE_EXISTS;
167.   }
168.
169.   public void drawPage(Graphics2D g2, PageFormat pf, int page)
170.   {
171.      if (message.equals("")) return;
172.      page--; // Учет титульной страницы.
173.
174.      drawCropMarks(g2, pf);
175.      g2.clip(new Rectangle2D.Double(0, 0, pf.getImageableWidth(),
176.         pf.getImageableHeight()));
177.      g2.translate(-page * pf.getImageableWidth(), 0);
178.      g2.scale(scale, scale);
179.      FontRenderContext context = g2.getFontRenderContext();
180.      Font f = new Font("Serif", Font.PLAIN, 72);
181.      TextLayout layout = new TextLayout(message, f, context);
182.      AffineTransform transform =
183.         AffineTransform.getTranslateInstance(0,
184.         layout.getAscent());
185.      Shape outline = layout.getOutline(transform);
186.      g2.draw(outline);
187.   }
188.
189.   /**
190.    * Отображение маркеров, отмечающих область, доступную
191.    * для печати.
192.    * @param g2 Графический контекст
193.    * @param pf Формат страницы
194.    */
```

```
195.    public void drawCropMarks(Graphics2D g2, PageFormat pf)
196.    {
197.       final double C = 36; // Размер маркера = 1/2 дюйма
198.       double w = pf.getImageableWidth();
199.       double h = pf.getImageableHeight();
200.       g2.draw(new Line2D.Double(0, 0, 0, C));
201.       g2.draw(new Line2D.Double(0, 0, C, 0));
202.       g2.draw(new Line2D.Double(w, 0, w, C));
203.       g2.draw(new Line2D.Double(w, 0, w - C, 0));
204.       g2.draw(new Line2D.Double(0, h, 0, h - C));
205.       g2.draw(new Line2D.Double(0, h, C, h));
206.       g2.draw(new Line2D.Double(w, h, w, h - C));
207.       g2.draw(new Line2D.Double(w, h, w - C, h));
208.    }
209.
210.    private String message;
211.    private double scale;
212. }
213.
214. /**
215.  * Титульная страница с заголовком.
216.  */
217. class CoverPage implements Printable
218. {
219.    /**
220.     * Создание титульной страницы.
221.     * @param t Заголовок
222.     */
223.    public CoverPage(String t)
224.    {
225.       title = t;
226.    }
227.
228.    public int print(Graphics g, PageFormat pf, int page)
229.       throws PrinterException
230.    {
231.       if (page >= 1) return Printable.NO_SUCH_PAGE;
232.       Graphics2D g2 = (Graphics2D)g;
233.       g2.setPaint(Color.black);
234.       g2.translate(pf.getImageableX(), pf.getImageableY());
235.       FontRenderContext context = g2.getFontRenderContext();
236.       Font f = g2.getFont();
237.       TextLayout layout = new TextLayout(title, f, context);
238.       float ascent = layout.getAscent();
239.       g2.drawString(title, 0, ascent);
240.       return Printable.PAGE_EXISTS;
241.    }
242.
243.    private String title;
244. }
245.
246. /**
247.  * Данный класс реализует универсальное диалоговое
248.  * окно предварительного просмотра.
249.  */
250. class PrintPreviewDialog extends JDialog
251. {
```

```
252.    /**
253.     * Создание диалогового окна предварительного просмотра.
254.     * @param p Объект Printable
255.     * @param pf Формат страницы
256.     * @param pages Количество страниц в p
257.     */
258.    public PrintPreviewDialog(Printable p, PageFormat pf,
259.       int pages)
260.    {
261.       Book book = new Book();
262.       book.append(p, pf, pages);
263.       layoutUI(book);
264.    }
265.
266.    /**
267.     * Создание диалогового окна предварительного просмотра.
268.     * @param b Объект Book
269.     */
270.    public PrintPreviewDialog(Book b)
271.    {
272.       layoutUI(b);
273.    }
274.
275.    /**
276.     * Компоновка интерфейсных элементов в диалоговом
277.     * окне предварительного просмотра.
278.     * @param book Объект Book, предназначенный для просмотра
279.     */
280.    public void layoutUI(Book book)
281.    {
282.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
283.
284.       canvas = new PrintPreviewCanvas(book);
285.       add(canvas, BorderLayout.CENTER);
286.
287.       JPanel buttonPanel = new JPanel();
288.
289.       JButton nextButton = new JButton("Next");
290.       buttonPanel.add(nextButton);
291.       nextButton.addActionListener(new ActionListener()
292.          {
293.             public void actionPerformed(ActionEvent event)
294.             {
295.                canvas.flipPage(1);
296.             }
297.          });
298.
299.       JButton previousButton = new JButton("Previous");
300.       buttonPanel.add(previousButton);
301.       previousButton.addActionListener(new ActionListener()
302.          {
303.             public void actionPerformed(ActionEvent event)
304.             {
305.                canvas.flipPage(-1);
306.             }
307.          });
308.
```

```
309.      JButton closeButton = new JButton("Close");
310.      buttonPanel.add(closeButton);
311.      closeButton.addActionListener(new ActionListener()
312.        {
313.          public void actionPerformed(ActionEvent event)
314.          {
315.            setVisible(false);
316.          }
317.        });
318.      add(buttonPanel, BorderLayout.SOUTH);
319.    }
320.
321.    private PrintPreviewCanvas canvas;
322.
323.    private static final int DEFAULT_WIDTH = 300;
324.    private static final int DEFAULT_HEIGHT = 300;
325. }
326. /**
327.  * Холст для предварительного просмотра.
328.  */
329. class PrintPreviewCanvas extends JPanel
330. {
331.    /**
332.     * Создание холста для предварительного просмотра.
333.     * @param b Объект Book, предназначенный для просмотра
334.     */
335.    public PrintPreviewCanvas(Book b)
336.    {
337.      book = b;
338.      currentPage = 0;
339.    }
340.    public void paintComponent(Graphics g)
341.    {
342.      Graphics2D g2 = (Graphics2D)g;
343.      PageFormat pageFormat = book.getPageFormat(currentPage);
344.
345.      double xoff;    // Смещение начала страницы в окне по оси x.
346.      double yoff;    // Смещение начала страницы в окне по оси y.
347.      double scale;   // Коэффициент масштабирования для приведения
348.                      // размеров отображаемой страницы
349.                      // в соответствие с размерами окна.
350.      double px = pageFormat.getWidth();
351.      double py = pageFormat.getHeight();
352.      double sx = getWidth() - 1;
353.      double sy = getHeight() - 1;
354.      if (px / py < sx / sy) // Центрирование по горизонтали.
355.      {
356.        scale = sy / py;
357.        xoff = 0.5 * (sx - scale * px);
358.        yoff = 0;
359.      }
360.      else // Центрирование по вертикали.
361.      {
362.        scale = sx / px;
363.        xoff = 0;
364.        yoff = 0.5 * (sy - scale * py);
365.      }
```

```
366.      g2.translate((float)xoff, (float)yoff);
367.      g2.scale((float)scale, (float)scale);
368.
369.      // Отображение контуров страницы (поля игнорируются).
370.      Rectangle2D page = new Rectangle2D.Double(0, 0, px, py);
371.      g2.setPaint(Color.white);
372.      g2.fill(page);
373.      g2.setPaint(Color.black);
374.      g2.draw(page);
375.
376.      Printable printable = book.getPrintable(currentPage);
377.      try
378.      {
379.        printable.print(g2, pageFormat, currentPage);
380.      }
381.      catch (PrinterException e)
382.      {
383.        g2.draw(new Line2D.Double(0, 0, px, py));
384.        g2.draw(new Line2D.Double(px, 0, 0, py));
385.      }
386.    }
387.
388.    /**
389.     * Переход к странице с заданным номером.
390.     * @param by Число страниц для перелистывания. Отрицательное
391.     * значение соответствует перелистыванию назад
392.     */
393.    public void flipPage(int by)
394.    {
395.      int newPage = currentPage + by;
396.      if (0 <= newPage && newPage < book.getNumberOfPages())
397.      {
398.        currentPage = newPage;
399.        repaint();
400.      }
401.    }
402.
403.    private Book book;
404.    private int currentPage;
405. }
```

API java.awt.print.PrinterJob 1.2

- `void setPageable(Pageable p)`

  Задает для печати объект типа `Pageable`, например, объект `Book`.

API java.awt.print.Book 1.2

- `void append(Printable p, PageFormat format)`
- `void append(Printable p, PageFormat format, int pageCount)`

  Добавляет в данную книгу новый раздел. Если счетчик страниц не задан, добавляется первая страница.

- `Printable getPrintable(int page)`

  Возвращает объект `Printable` для указанной страницы.

Службы печати

В предыдущих разделах рассматривались способы печати двухмерной графики. Однако интерфейс печати в Java SE 1.4 обладает более широкими возможностями. В API определены многие другие типы данных и обеспечивается поиск служб печати для их вывода. Некоторые из поддерживаемых типов данных перечислены ниже.

- Изображения в форматах GIF, JPEG или PNG.
- Текстовые документы в форматах HTML, PostScript или PDF.
- Неформатированные данные принтера.
- Экземпляры класса, который реализуют интерфейсы `Printable`, `Pageable` или `RenderableImage`.

Сами по себе данные могут храниться в любом источнике байтов или символов, например, во входном потоке, по URL-адресу или в массиве. Сочетание типа данных и источника данных называется *типом документа* (document flavor). Класс `DocFlavor` содержит несколько внутренних классов для различных источников данных. Каждый из этих внутренних классов содержит константы для указания типа документа. Например, показанная ниже константа описывает изображение в формате GIF, который считывается из входного потока:

```
DocFlavor.INPUT_STREAM.GIF
```

Все другие типы документов, т.е. допустимые сочетания источника и типа данных (с указанием MIME-типа), перечислены в табл. 7.3.

Таблица 7.3. Типы документов, определяемые для служб печати

Источник данных	Тип данных	MIME-тип
`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` (с кодировкой, специфической 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` (порядок следования байтов little-endian)
	`TEXT_HTML_UTF_16BE`	`text/html; charset=utf-16be` (порядок следования байтов big-endian)
	`TEXT_PLAIN_HOST`	`text/plain` (с кодировкой, специфической для конкретного узла)
	`TEXT_PLAIN_US_ASCII`	`text/plain; charset=us-ascii`

Окончание табл. 7.3

Источник данных	Тип данных	MIME-тип
	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 (порядок следования байтов little-endian)
	TEXT_PLAIN_UTF_16BE	text/plain; charset=utf-16be (порядок следования байтов big-endian)
	PCL	application/vnd.hp-PCL (Hewlett Packard Printer Control Language)
	AUTOSENSE	application/octet-stream (неформатированные данные принтера)
READER	TEXT_HTML	text/html; charset=utf-16
STRING	TEXT_PLAIN	text/plain; charset=utf-16
CHAR_ARRAY		
SERVICE_FORMATTED	PRINTABLE	—
	PAGEABLE	—
	RENDERABLE_IMAGE	—

Предположим, что нужно напечатать изображение в формате GIF, которое находится в файле. Для выполнения этой задачи следует определить, имеется ли соответствующая *служба печати* (print service), которая способна управлять выполнением данной задачи. Статический метод `lookupPrintServices()` класса `PrintServiceLookup` возвращает массив объектов `PrintService`, которые могут управлять печатью определенных типов документов:

```
DocFlavor flavor = DocFlavor.INPUT_STREAM.GIF;
PrintService[] services = PrintServiceLookup.lookupPrintServices(flavor, null);
```

Второй параметр метода `lookupPrintServices()` имеет значение `null`, которое означает, что поиск подходящей службы не ограничивается какими-то атрибутами печати. Более подробно атрибуты печати рассматриваются в следующем разделе.



В пакете Java SE 6 поддерживаются базовые службы печати для основных типов документов, таких как изображения и двухмерная графика. Но при попытке печати текстового документа или HTML-документа будет возвращен пустой массив.

Если в результате поиска служб печати получен массив `PrintService[]` с более чем одним элементом, то среди перечисленных служб печати следует выбирать какую-то одну. Для извлечения списка имен служб (например, имен принтеров) и предоставления пользователю возможности выбора можно вызвать метод `getName()` класса `printService`.

Далее нужно получить задание печати выбранной службы:

```
DocPrintJob job = services[i].createPrintJob();
```

Для печати объекта с заданным типом документа необходимо создать объект, который реализует интерфейс `Doc`. Для этого в библиотеке Java предусмотрен класс

SimpleDoc. Конструктору SimpleDoc следует передать объект — источник данных, тип документа и необязательный набор атрибутов.

```
InputStream in = new FileInputStream(fileName);
Doc doc = new SimpleDoc(in, flavor, null);
```

Теперь все готово для выполнения печати:

```
job.print(doc, null);
```

Как и раньше, значение null второго параметра можно заменить набором атрибутов.

Обратите внимание, что описанный процесс печати совершенно отличается от того, который рассматривался в предыдущем разделе, потому что здесь отсутствует какое-либо взаимодействие пользователя и процесса печати посредством диалоговых окон. На основе данного механизма можно реализовать процесс печати на сервере; при этом пользователи должны указывать задания посредством Web-форм.

Программа, код которой приведен в листинге 7.10, демонстрирует использование службы печати для вывода графического файла.

Листинг 7.10. Содержимое файла PrintServiceTest.java

```
1.  import java.io.*;
2.  import javax.print.*;

3.  /**
4.   * Пример использования служб печати.
5.   * Программа позволяет вывести изображение
6.   * GIF,используя любые службы печати, которые
7.   * поддерживают данный тип документа.
8.   * @version 1.10 2007-08-16
9.   * @author Cay Horstmann
10.  */
11. public class PrintServiceTest
12. {
13.    public static void main(String[] args)
14.    {
15.       DocFlavor flavor = DocFlavor.URL.GIF;
16.       PrintService[] services = PrintService
17.            Lookup.lookupPrintServices(flavor, null);
18.       if (args.length == 0)
19.       {
20.        if (services.length == 0) System.out.println("No printer for flavor " + flavor);
21.        else
22.        {
23.          System.out.println("Specify a file of flavor " + flavor
24.            + "\nand optionally the number of the desired printer.");
25.          for (int i = 0; i < services.length; i++)
26.            System.out.println((i + 1) + ": " + services[i].getName());
27.        }
28.        System.exit(0);
29.       }
30.       String fileName = args[0];
31.       int p = 1;
32.       if (args.length > 1) p = Integer.parseInt(args[1]);
33.       try
34.       {
35.          if (fileName == null) return;
36.          FileInputStream in = new FileInputStream(fileName);
```

```
37.         Doc doc = new SimpleDoc(in, flavor, null);
38.         DocPrintJob job = services[p - 1].createPrintJob();
39.        job.print(doc, null);
40.       }
41.      catch (FileNotFoundException e)
42.      {
43.        e.printStackTrace();
44.      }
45.      catch (PrintException e)
46.      {
47.        e.printStackTrace();
48.       }
49.     }
50.  }
```

API javax.print.PrintServiceLookup 1.4

- `PrintService[] lookupPrintServices(DocFlavor flavor, AttributeSet attributes)`

  Ищет службы печати, которые могут управлять заданным типом документа и атрибутами.

  Параметры: `flavor` Тип документа
  `attributes` Требуемые атрибуты печати или значение `null`, если атрибуты не принимаются в расчет

API javax.print.PrintService 1.4

- `DocPrintJob createPrintJob()`

  Создает задание печати для вывода экземпляра класса, реализующего интерфейс `Doc` (например, `SimpleDoc`).

API javax.print.DocPrintJob 1.4

- `void print(Doc doc, PrintRequestAttributeSet attributes)`

  Выводит на печать заданный документ с указанными атрибутами.

  Параметры: `doc` Документ
  `attributes` Атрибуты печати или значение `null`, если атрибуты печати не требуются

API javax.print.SimpleDoc 1.4

- `SimpleDoc(Object data, DocFlavor flavor, DocAttributeSet attributes)`

  Создает объект `SimpleDoc`, который может быть выведен на печать с помощью `DocPrintJob`.

  Параметры: `data` Объект, который содержит данные для печати, например, входной поток или объект `Printable`
  `flavor` Тип документа с печатаемыми данными
  `attributes` Атрибуты документа или значение `null`, если атрибуты печати не требуются

Потоковые службы печати

Обычные службы печати посылают данные принтеру. Потоковые службы печати создают такие же данные, но посылают их в поток. Это бывает необходимо для задержки печати или для использования информации в других программах. Например, если данные печати имеют формат PostScript, то полезно сохранить их в файле, поскольку многие программы могут работать с файлами этого типа. В Java предусмотрены потоковые службы, которые могут генерировать изображения и двухмерную графику в формате PostScript. Эти службы можно использовать во всех системах, даже в тех, у которых нет локальных принтеров.

Перечисление потоковых служб печати организовано немного сложнее, чем перечисление локальных служб печати. Для этого необходимо получить `DocFlavor` для печатаемого объекта и MIME-тип выходного потока, а затем получить массив `StreamPrintServiceFactory`.

```
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 одну фабрику потоковой службы от другой, так что в данном случае следует выбрать самую первую, т.е. `factories[0]`. Для получения объекта-службы печати `StreamPrintService` нужно вызвать метод `getPrintService()` с указанием выходного потока в качестве параметра.

```
OutputStream out = new FileOutputStream(fileName);
StreamPrintService service = factories[0].getPrintService(out);
```

Класс `StreamPrintService` является подклассом класса `PrintService`, поэтому для организации процесса печати потребуется лишь выполнить остальные действия, описанные в предыдущем разделе.

API javax.print.StreamPrintServiceFactory 1.4

- `StreamPrintServiceFactory[] lookupStreamPrintServiceFactories (DocFlavor flavor, String mimeType)`

  Ищет потоковые службы печати, которые могут выводить указанный тип документа и создавать выходной поток заданного MIME-типа.

- `StreamPrintService getPrintService(OutputStream out)`

  Возвращает службу печати, которая передает данные в указанный выходной поток.

Атрибуты печати

API службы печати содержит сложный набор интерфейсов и классов для указания различных типов атрибутов, которые делятся на четыре основные группы. Первые две группы атрибутов определяют характеристики печатаемых документов.

- *Атрибуты запроса печати* (print request attribute) запрашивают отдельные характеристики всех документов данного задания печати. В качестве примеров таких характеристик можно привести двухстороннюю печать или размер бумаги.
- *Атрибуты документа* (doc attributes) используются только для какого-то одного объекта-документа.

Оставшиеся две группы атрибутов содержат информацию о принтере и состоянии задания печати.

- *Атрибуты службы печати* (print service attribute) содержат информацию о службе печати (например, о производителе и модели принтера, о том, принял принтер в данный момент задание печати или нет).
- *Атрибуты задания печати* (print job attribute) содержат информацию о состоянии определенного задания печати (например, выполнено уже задание или нет).

Для описания различных атрибутов предусмотрен интерфейс `Attribute` и перечисленные ниже дочерние 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 материала. Этот класс реализует интерфейсы `PrintRequestAttribute` и `PrintJobAttribute`. Задание печати может содержать запрос на печать нескольких копий, а атрибутом задания печати может быть количество фактически напечатанных копий. Фактическое количество копий может быть меньше требуемого количества из-за ограничений принтера или отсутствия бумаги.

Интерфейс `SupportedValuesAttribute` показывает, что значения атрибута не отображают реальный запрос или данные о текущем состоянии, а демонстрируют возможности службы. Например, объект класса `CopiesSupported` реализует интерфейс `SupportedValuesAttribute` и сообщает, что принтер поддерживает печать от 1 до 99 копий.

На рис. 7.38 показана диаграмма классов, которые образуют иерархию атрибутов.

В дополнение к интерфейсам и классам для отдельных атрибутов, в API службы печати предусмотрены интерфейсы и классы для наборов атрибутов.

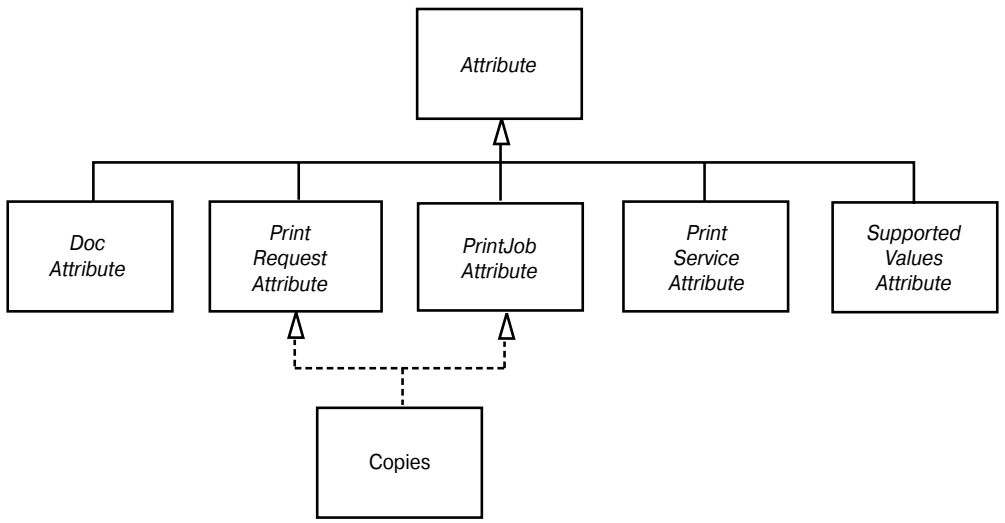


Рис. 7.38. Иерархия атрибутов

Например, интерфейс `AttributeSet` содержит четыре дочерних интерфейса:

```
PrintRequestAttributeSet
DocAttributeSet
PrintServiceAttributeSet
PrintJobAttributeSet
```

Для каждого из них предусмотрен класс, реализующий этот интерфейс:

```
HashAttributeSet
HashPrintRequestAttributeSet
HashDocAttributeSet
HashPrintServiceAttributeSet
HashPrintJobAttributeSet
```

На рис. 7.39 показана диаграмма классов, которые образуют иерархию наборов атрибутов. Ниже показан типичный пример создания набора атрибутов для запроса на печать:

```
PrintRequestAttributeSet attributes = new HashPrintRequestAttributeSet();
```

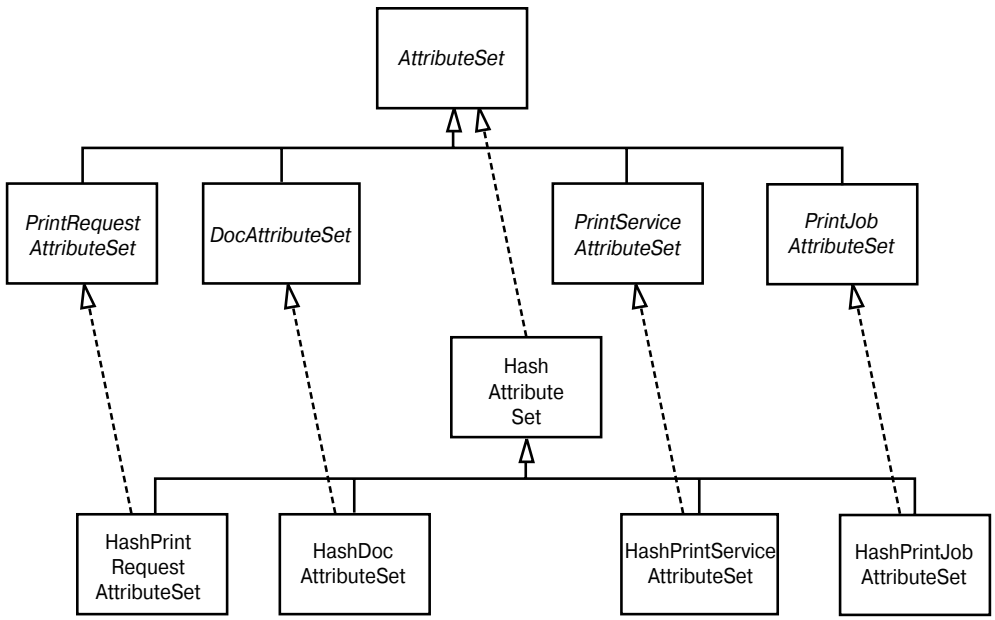


Рис. 7.39. Иерархия наборов атрибутов

После создания набора вам нет необходимости принимать во внимание префикс `Hash`.

Зачем же нужны все эти интерфейсы? Они дают возможность проверить корректность использования атрибутов. Например, интерфейс `DocAttributeSet` принимает только те объекты, которые реализуют интерфейс `DocAttribute`, а попытка добавить другой атрибут приведет к ошибке.

Набор атрибутов представляет собой специализированную карту, в которой ключи являются объектами класса `Class`, а значения — объектами класса, который реализует интерфейс `Attribute`. Например, если создать и вставить объект `Copies(10)` в набор атрибутов, то его ключом будет объект `Copies.class` класса `Class`. Этот ключ

называется *категорией* (category) атрибутов. Интерфейс `Attribute` содержит объявление показанного ниже метода, который возвращает категорию атрибута:

```
Class getCategory()
```

Класс `Copies` содержит определение этого метода для возврата объекта `Copies.class`, но вовсе не обязательно, чтобы категория была объектом того же класса, что и класс атрибута.

При добавлении атрибута в набор атрибутов категория извлекается автоматически, поэтому достаточно добавить значение атрибута:

```
attributes.add(new Copies(10));
```

Если впоследствии добавить другой атрибут из той же категории, то второй атрибут будет записан вместо первого.

Категория используется как ключ для извлечения атрибута:

```
AttributeSet attributes job.getAttributes();
Copies copies = (Copies)attribute.get(Copies.class);
```

Атрибуты также группируются по типам значений. Например, атрибут `Copies` может иметь любое целочисленное значение, поскольку класс `Copies` является расширением класса `IntegerSyntax`, который предназначен для обработки всех целочисленных атрибутов. Метод `getValue()` возвращает целочисленное значение атрибута:

```
int n = copies.getValue();
```

Перечисленные ниже классы инкапсулируют соответственно строку, дату/время и URI:

```
TextSyntax
DateTimeSyntax
URISyntax
```

Наконец, есть много атрибутов, которые могут принимать ограниченное количество значений. Например, атрибут `PrintQuality` имеет всего три допустимых значения для указания экономного, нормального и высокого качества печати, которые представлены следующими константами:

```
PrintQuality.DRAFT
PrintQuality.NORMAL
PrintQuality.HIGH
```

Классы атрибутов с ограниченным количеством значений расширяют класс `EnumSyntax`, который предоставляет удобные методы для безопасной установки значений перечислений. Благодаря наличию таких классов разработчику при использовании атрибутов не нужно беспокоиться о деталях. Достаточно добавить в набор атрибутов именованные значения:

```
attributes.add(PrintQuality.HIGH);
```

Для проверки значения атрибута нужно применить следующую конструкцию:

```
if (attributes.get(PrintQuality.class) == PrintQuality.High)
...
```

В табл. 7.4 представлены атрибуты печати. Во втором столбце перечислены суперклассы класса атрибутов (например, класс `IntegerSyntax` для атрибута `Copies`) или набор значений. В последних четырех столбцах отмечены интерфейсы, которые реализует данный класс атрибута: `DocAttribute` (DA), `PrintJobAttribute` (PJA), `PrintRequestAttribute` (PRA) и `PrintServiceAttribute` (PSA).



Как видите, многие из перечисленных в табл. 7.4 атрибутов являются узкоспециализированными и появились вследствие применения протокола Internet Printing Protocol 1.1 (RFC 2911).



В более ранней версии API печати предлагались классы `JobAttributes` и `PageAttributes`, которые служили той же цели, что и описываемые в этом разделе атрибуты печати. Сейчас эти классы уже считаются устаревшими.

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`		×	×	
`DataTimeAtCompleted`	`DateTimeSyntax`		×		
`DataTimeAtCreation`	`DateTimeSyntax`		×		
`DataTimeAtProcessing`	`DateTimeSyntax`		×		
`Destination`	`URISyntax`		×	×	
`DocumentName`	`TextSyntax`	×			
`Fidelity`	`FIDELITY_TRUE`, `FIDELITY_FALSE`		×	×	
`Finishing`	`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`	`STANDART`, `NONE`		×	×	

Продолжение табл. 7.4

Атрибут	Суперкласс или набор констант	DA	PJA	PRA	PSA
JobState	ABORTED, CANCELED, COMPLETED, PENDING, PENDING_HELD, PROCESSING, PROCESSING_STOPPED		×		
JobStateReason	ABORTED_BY_SYSTEM, DOCUMENT_FORMAT_ERROR и т.д.				
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, а также другие размеры страниц и конвертов				
MediaSizeName	ISO_A0 — ISO_A10, ISO_B0 — ISO_B10, ISO_C0 — ISO_C10, NA_LETTER, NA_LEGAL, а также другие имена форматов страниц и конвертов	×	×	×	
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	×	×	×	

Окончание табл. 7.4

Атрибут	Суперкласс или набор констант	DA	PJA	PRA	PSA
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				×
PrinterMessageFromOperator	TextSyntax				×
PrinterMoreInfo	URISyntax				×
PrinterMoreInfoManufacturer	URISyntax				×
PrinterName	TextSyntax				×
PrinterResolution	ResolutionSyntax	×	×	×	
PrinterState	PROCESSING, IDLE, STOPPED, UNKNOWN				×
PrinterStateReason	COVER_OPEN, FUSER_OVER_TEMP, MEDIA_JAM и др.				
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 (TWO_SIDED_LONG_EDGE), TUMBLE (TWO_SIDED_SHORT_EDGE)	×	×	×	

API javax.print.attribute.Attribute 1.4

- `Class getCategory()`

  Возвращает категорию атрибута.

- `String getName()`

  Возвращает имя атрибута.

API javax.print.attribute.AttributeSet 1.4

- `boolean add(Attribute attr)`

  Добавляет атрибут в набор атрибутов. Если набор имеет другой атрибут из той же категории, то новый атрибут замещает его. Возвращает значение `true`, если в результате выполнения операции набор изменился.

- `Attribute get(Class category)`

  Возвращает атрибут с ключом данной категории или значение `null`, если такого атрибута не существует.

- `boolean remove(Attribute attr)`
- `boolean remove(Class category)`

  Удаляет указанный атрибут или атрибут указанной категории из набора атрибутов. Этот метод возвращает значение `true`, если в результате выполнения операции набор изменился.

- `Attribute[] toArray()`

  Возвращает массив, содержащий все атрибуты данного набора.

API javax.print.PrintService 1.4

- `PrintServiceAttributeSet getAttributes()`

  Возвращает атрибуты данной службы печати.

API javax.print.DocPrintJob 1.4

- `PrintJobAttributeSet getAttributes()`

  Возвращает атрибуты данного задания печати.

Итак, обсуждение средств печати завершено. Теперь вы знаете, как выводить двухмерную графику и другие типы документов, как организовывать перечисление принтеров и потоковых служб печати и как устанавливать и извлекать атрибуты. Теперь перейдем к рассмотрению двух вопросов, чрезвычайно важных для создания пользовательских интерфейсов: организация буфера обмена и поддержка перетаскивания объектов.

Буфер обмена

Одним из наиболее полезных и удобных механизмов пользовательского интерфейса в средах с графическим пользовательским интерфейсом (вроде Windows и X Window System) является механизм *вырезания и вставки* (cut and paste). Он позволяет выбирать какие-то данные в одной программе, вырезать или копировать их в буфер обмена, а потом переходить в другую программу и вставлять в ней содержимое буфе-

ра. 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 привычен, что большинство пользователей применяют его, совершенно не задумываясь.

Хотя действия, необходимые для работы с буфером обмена, очень просты, реализация буфера гораздо сложнее, чем это может показаться на первый взгляд. Предположим, что какой-то фрагмент текста скопирован из текстового редактора в буфер обмена. При вставке этого текста в другой текстовый редактор было бы желательно, чтобы шрифт и элементы форматирования данного фрагмента оставались прежними. Это значит, что буфер обмена должен также содержать информацию о форматировании. А при вставке этого фрагмента текста в поле редактирования достаточно вставить только текст. Для обеспечения такой гибкости данных нужно иметь возможность предоставлять данные в различных форматах, а потребитель данных должен уметь выбирать один из них.

Системный буфер обмена в Microsoft Windows и Macintosh реализован практически одинаково; эти реализации различаются между собой лишь деталями. Например, 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 при работе с программой, рассматриваемой в настоящем разделе.



Для проверки типов объектов, которые могут передаваться между приложениями Java и системным буфером обмена, следует обратиться к файлу `jre/lib/flavormap.properties` на используемой платформе.

Часто в программах требуется организовать поддержку копирования и вставки таких типов данных, которые не поддерживаются системным буфером обмена. API передачи данных поддерживает ссылки на произвольные локальные объекты в рамках одной виртуальной машины. Между различными виртуальными машинами можно передавать сериализируемые объекты и ссылки на удаленные объекты.

В табл. 7.5 перечислены допустимые способы передачи данных с помощью буфера обмена.

Таблица 7.5. Возможности механизма передачи данных Java

Способ передачи данных	Формат данных
Между программой на языке Java и платформенно-ориентированной программой	Текст, рисунки, списки и т.п. (в зависимости от платформы)
Между двумя программами на языке Java	Сериализируемые и удаленные объекты
Внутри одной Java-программы	Любые объекты

Классы и интерфейсы для передачи данных

Передача данных в технологии Java реализована посредством пакета `java.awt.datatransfer`. Ниже перечислены некоторые особенности классов и интерфейсов из данного пакета.

- Объекты, переносимые посредством буфера обмена, должны реализовывать интерфейс `Transferable`.
- Класс `Clipboard` описывает буфер обмена. Системный буфер обмена является конкретным примером реализации класса `Clipboard`.

- Класс DataFlavor описывает типы документов, которые могут быть помещены в буфер обмена.
- Класс StringSelection реализует интерфейс Transferable и используется для передачи строк.
- Для получения уведомления о перезаписи содержимого буфера класс должен реализовывать интерфейс ClipboardOwner. Управление правом владения интерфейсом позволяет выполнять "отложенное форматирование" сложных данных. Если программа передает простые данные (например, 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 представлены различными типами документов, то не рекомендуется обрабатывать все эти типы документов, так как многие из них могут вообще не понадобиться. Однако нужно сохранить данные в буфере таким образом, чтобы впоследствии можно было обрабатывать типы документов по мере необходимости. Поэтому владелец буфера обмена с помощью метода lostOwnership() должен быть своевременно уведомлен об изменении содержимого. Изменение содержимого буфера означает, что информация о сложных данных больше не требуется. В примерах данной главы механизм управления правами владения буфером обмена не используется.

Передача текста

Для первого знакомства с классами передачи данных начнем с простейшей ситуации, т.е. передачи текста с помощью системного буфера обмена. Сначала нужно получить ссылку на системный буфер обмена:

```
Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
```

Строки, которые передаются в буфер обмена, необходимо заключить в объект-оболочку StringSelection с помощью показанного ниже конструктора:

```
String text = ...
StringSelection selection = new StringSelection(text);
```

Передача осуществляется при вызове метода setContents(), который принимает объекты StringSelection и ClipBoardOwner в качестве параметров. Если владелец буфера обмена не имеет значения, то второму параметру следует присвоить значение null:

```
clipboard.setContents(selection, null);
```

Ниже показано, как можно выполнить обратную операцию, то есть считать строку из буфера обмена:

```
DataFlavor flavor = DataFlavor.stringFlavor;
if (clipboard.isDataFlavorAvailable(flavor)
   String text = (String) clipboard.getData(flavor);
```

Методу getContents() в качестве параметра вообще-то должна передаваться ссылка Object, указывающая на запрашиваемый объект, но поскольку в текущей реализации класса Clipboard это игнорируется, в демонстрируемой здесь программе передается просто значение null.

Возвращаемым значением метода getContents() тоже может null, означающее, что либо буфер обмена пуст, либо не содержит таких данных, которые платформа Java знает, как можно извлечь в виде текста.

В листинге 7.11 приведен код программы, которая демонстрирует механизм выполнения копирования и вставки в приложениях Java с использованием системного буфера обмена. При выборе какого-то текста в текстовой области и выполнении щелчка на кнопке Copy (Копировать) в этой программе, выбранный текст копируется в системный буфер обмена, после чего его можно вставлять в окно любого текстового редактора (рис. 7.40). Помимо этого, еще также можно выполнять и обратную операцию, то есть копировать текст из окна любого текстового редактора и вставлять его в эту демонстрационную программу путем выполнения в ней щелчка на кнопке Paste (Вставить).

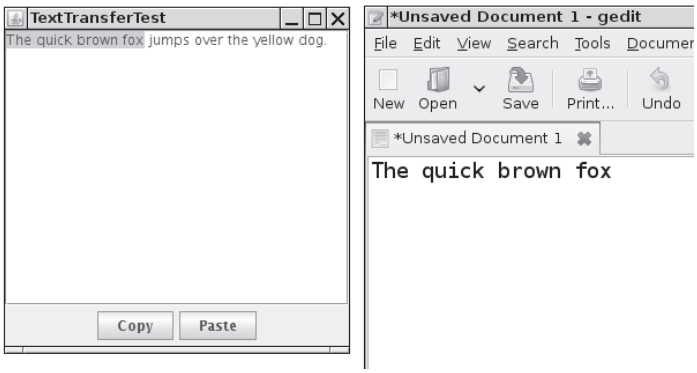


Рис. 7.40. Программа `TextTransferTest`

Листинг 7.11. Содержимое файла `TextTransferTest.java`

```
1.  import java.awt.*;
2.  import java.awt.datatransfer.*;
3.  import java.awt.event.*;
4.  import java.io.*;
5.
6.  import javax.swing.*;
7.
8.  /**
9.   * Пример передачи текста из Java-приложения
10.  * в системный буфер обмена.
11.  * @version 1.13 2007-08-16
12.  * @author Cay Horstmann
13.  */
14. public class TextTransferTest
15. {
16.    public static void main(String[] args)
17.    {
18.      EventQueue.invokeLater(new Runnable()
19.       {
20.         public void run()
21.         {
22.            JFrame frame = new TextTransferFrame();
23.            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24.            frame.setVisible(true);
25.          }
26.       });
27.    }
28. }
```

```
29. /**
30.  * Фрейм, содержащий текстовую область и кнопки
31.  * для копирования и вставки текста.
32.  */
33. class TextTransferFrame extends JFrame
34. {
35.    public TextTransferFrame()
36.    {
37.       setTitle("TextTransferTest");
38.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
39.
40.       textArea = new JTextArea();
41.       add(new JScrollPane(textArea), BorderLayout.CENTER);
42.       JPanel panel = new JPanel();
43.
44.       JButton copyButton = new JButton("Copy");
45.       panel.add(copyButton);
46.       copyButton.addActionListener(new ActionListener()
47.          {
48.             public void actionPerformed(ActionEvent event)
49.             {
50.                copy();
51.             }
52.          });
53.
54.       JButton pasteButton = new JButton("Paste");
55.       panel.add(pasteButton);
56.       pasteButton.addActionListener(new ActionListener()
57.          {
58.             public void actionPerformed(ActionEvent event)
59.             {
60.                paste();
61.             }
62.          });
63.
64.       add(panel, BorderLayout.SOUTH);
65.    }
66.
67.    /**
68.     * Копирование выделенного текста
69.     * в системный буфер обмена.
70.     */
71.    private void copy()
72.    {
73.       Clipboard clipboard =
74.          Toolkit.getDefaultToolkit().getSystemClipboard();
75.       String text = textArea.getSelectedText();
76.       if (text == null) text = textArea.getText();
77.       StringSelection selection = new StringSelection(text);
78.       clipboard.setContents(selection, null);
79.    }
80.
81.    /**
82.     * Вставка текста из системного буфера
83.     * обмена в текстовую область.
84.     */
```

```
85.    private void paste()
86.    {
87.       Clipboard clipboard =
88.          Toolkit.getDefaultToolkit().getSystemClipboard();
89.       DataFlavor flavor = DataFlavor.stringFlavor;
90.       if (clipboard.isDataFlavorAvailable(flavor))
91.       {
92.          try
93.          {
94.             String text = (String) clipboard.getData(flavor);
95.             textArea.replaceSelection(text);
96.          }
97.          catch (UnsupportedFlavorException e)
98.          {
99.             JOptionPane.showMessageDialog(this, e);
100.         }
101.         catch (IOException e)
102.         {
103.            JOptionPane.showMessageDialog(this, e);
104.         }
105.      }
106.   }
107.
108.   private JTextArea textArea;
109.
110.   private static final int DEFAULT_WIDTH = 300;
111.   private static final int DEFAULT_HEIGHT = 300;
112. }
```

API java.awt.Toolkit 1.0

- `Clipboard getSystemClipboard()`
  Возвращает системный буфер обмена.

API java.awt.datatransfer.Clipboard 1.1

- `Transferable getContents(Object requester)`
  Возвращает содержимое буфера обмена.

  Параметр: `requester` Объект, который запрашивает содержимое буфера обмена; это значение пока не используется

- `Transferable getContents(Object requester)`
  Помещает данные в буфер обмена.

  Параметры: `contents` Объект, инкапсулирующий содержимое буфера обмена
  `owner` Объект, который должен быть уведомлен об изменении содержимого буфера обмена (уведомление выполняется с помощью метода `lostOwnership()`), либо значение `null`, если такое уведомление не требуется

- `boolean isDataFlavorAvailable(DataFlavor flavor)` 5.0
  Возвращает значение `true`, если в буфере обмена содержатся данные, соответствующие указанному типу.

- `Object getData(DataFlavor flavor)` 5.0

  Возвращает данные указанного типа либо, если они недоступны, генерирует исключение `UnsupportedFlavorException`.

API java.awt.datatransfer.ClipboardOwner 1.1

- `void lostOwnership (Clipboard clipboard, Transferable contents)`

  Уведомляет объект о том, что он больше не является владельцем содержимого буфера обмена.

  Параметры: `clipboard` Буфер обмена, содержимое которого изменилось
  `contents` Объект, который был помещен в буфер обмена владельцем

API java.awt.datatransfer.Transferable 1.1

- `boolean isDataFlavorSupported(DataFlavor flavor)`

  Возвращает значение `true`, если указанный тип документа поддерживается, и значение `false` в противном случае.

- `Object getTransferData(DataFlavor flavor)`

  Возвращает данные, форматированные в соответствии с указанным типом документа. Генерирует исключение `UnsupportedFlavorException`, если указанный тип документа не поддерживается.

Интерфейс Transferable и типы документов

Объект типа документа `DataFlavor` обладает следующими характеристиками:

- MIME-тип (например, `image/gif`);
- класс представления, который используется для доступа к данным (например, `java.awt.Image`).

Кроме того, каждый тип документа имеет название, ориентированное на восприятие пользователем (например, `"GIF Image"`).

Класс представления может быть указан с помощью параметра `class` в MIME-типе:

```
image/gif;class=java.awt.Image
```



Это всего лишь пример, который иллюстрирует синтаксис указания типа документа. Никакого стандартного типа для передачи графических данных в формате GIF не существует.

Если параметр `class` не задан, то в качестве класса представления используется `InputStream`.

Для передачи локальных, сериализированных и удаленных Java-объектов предусмотрены три MIME-типа:

```
applications/x-java-jvm-local-objectref
applications/x-java-serialized-object
applications/x-java-remote-object
```



Префикс `x-` означает экспериментальное имя, которое еще не утверждено IANA в качестве стандартного MIME-типа.

Например, стандартный тип `stringFlavor` описывается с помощью следующего MIME-типа:

```
applications/x-java-serialized-object;class=java.lang.String
```

У буфера обмена можно запрашивать и список всех доступных типов документов:

```
DataFlavor[] flavors = clipboard.getAvailableDataFlavors()
```

Кроме того, для буфера обмена можно еще также устанавливать обработчик `FlavorListener`, который будет уведомляться об изменении любого типа данных в буфере. Более подробную информацию о нем можно найти в приведенных ниже врезках API.

API java.awt.datatransfer.DataFlavor 1.1

- `DataFlavor(String mimeType, String humanPresentableName)`

  Создает тип документа, который описывает потоковые данные в формате, соответствующем указанному MIME-типу.

Параметры:	`mimeType`	MIME-тип
	`HumanPresentableName`	Имя MIME-типа, ориентированное на восприятие пользователем

- `DataFlavor(Class class, String humanPresentableName)`

  Создает тип документа, который описывает класс платформы Java с MIME-типом `applications/x-java-serialized-object;class=`*имя_класса*.

Параметры:	`class`	Класс, извлекаемый с помощью объекта `Transferable`
	`humanPresentableName`	Вариант имени, ориентированный на восприятие пользователем

- `String getMimeType()`

  Возвращает строку, представляющую MIME-тип.

- `boolean isMimeTypeEqual(String mimeType)`

  Проверяет соответствие данного типа документа указанному MIME-типу.

- `String humanPresentableName()`

  Возвращает имя формата для данного типа документа, ориентированное на восприятие пользователем.

- `Class getRepresentationClass()`

  Возвращает объект `Class`, представляющий класс объекта, который объект `Transferable` возвращает для данного типа документа. Это либо параметр `class` MIME-типа, либо класс `InputStream`.

API java.awt.datatransfer.Clipboard 1.1

- `DataFlavor[] getAvailableDataFlavors()` 5.0

  Возвращает массив доступных типов документа.

- `void addFlavorListener(FlavorListener listener)` 5.0
  Добавляет обработчик, который уведомляется об изменении набора доступных типов документа.

API java.awt.datatransfer.Transferable 1.1

- `DataFlavor[] getTransferDataFlavors()`
  Возвращает массив поддерживаемых типов документов.

API java.awt.datatransfer.FlavorListener 5.0

- `void flavorsChanged(FlavorEvent event)`
  Вызывается при изменении набора доступных типов документов.

Передача изображений через буфер обмена

Объекты, которые нужно передать через буфер обмена, должны реализовывать интерфейс `Transferable`. Класс `StringSelection` является единственным общедоступным классом в стандартной библиотеке Java, который реализует интерфейс `Transferable`. В этом разделе рассматривается способ передачи изображений через буфер обмена. В Java не предусмотрено никаких классов для такой передачи, поэтому их придется реализовать самостоятельно.

Создать такой класс несложно. Он лишь сообщает, что единственным доступным форматом данных является `DataFlavor.imageFlavor`, который содержит объект `image`.

```
class ImageTransferable implements Transferable
{
   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);
      }
   }
   private Image theImage;
}
```



Java SE содержит константу `DataFlavor.imageFlavor` и автоматически выполняет всю работу по преобразованию Java-изображений в платформенно-ориентировванные изображения буфера обмена. Но, как ни странно, в JDK не предусмотрено класса-оболочки, который необходим для помещения изображения в буфер обмена.

Программа, код которой приведен в листинге 7.12, демонстрирует передачу изображения между приложением Java и системным буфером обмена. При запуске она генерирует изображение, содержащее красный круг. Щелкните на кнопке Copy (Копировать) для копирования изображения в буфер обмена, а затем вставьте его в другое приложение, как показано на рис. 7.41. Далее скопируйте изображение из другого приложения в системный буфер обмена и затем щелкните на кнопке Paste (Вставить), чтобы вставить его обратно в исходную программу, как показано на рис. 7.42.

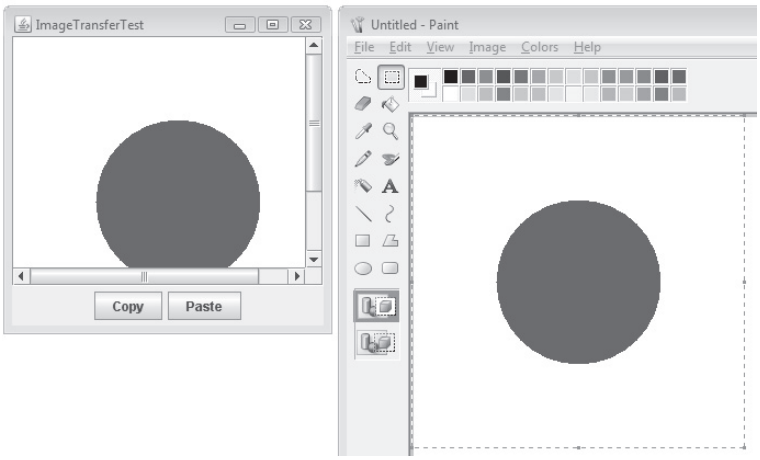


Рис. 7.41. Копирование изображения из Java-программы в платформенно-ориентированную программу



Рис. 7.42. Копирование изображения из платформенно-ориентированной программы в Java-программу

Данная программа является усовершенствованной версией предыдущей программы, предназначенной для передачи текста через буфер обмена. Здесь используется тип документа `DataFlavor.imageFlavor`, а для передачи изображения через системный буфер обмена применяется класс `ImageTransferable`.

Листинг 7.12. Содержимое файла `ImageTransferTest.java`

```
1.  import java.io.*;
2.  import java.awt.*;
3.  import java.awt.datatransfer.*;
4.  import java.awt.event.*;
5.  import java.awt.image.*;
6.  import javax.swing.*;
7.
8.  /**
9.   * Данная программа демонстрирует обмен изображениями
10.  * между Java-приложением и системным буфером обмена.
11.  * @version 1.22 2007-08-16
12.  * @author Cay Horstmann
13.  */
14. public class ImageTransferTest
15. {
16.    public static void main(String[] args)
17.    {
18.     EventQueue.invokeLater(new Runnable()
19.     {
20.       public void run()
21.       {
22.          JFrame frame = new ImageTransferFrame();
23.          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24.          frame.setVisible(true);
25.        }
26.      })
27.    }
28. }
29.
30. /**
31.  * Фрейм, содержащий изображение, представляемое с помощью метки,
32.  * а также кнопки для копирования и вставки изображения.
33.  */
34. class ImageTransferFrame extends JFrame
35. {
36.    public ImageTransferFrame()
37.    {
38.       setTitle("ImageTransferTest");
39.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
40.
41.       label = new JLabel();
42.       image = new BufferedImage(DEFAULT_WIDTH, DEFAULT_HEIGHT,
43.                                 BufferedImage.TYPE_INT_ARGB);
44.       Graphics g = image.getGraphics();
45.       g.setColor(Color.WHITE);
46.       g.fillRect(0, 0, DEFAULT_WIDTH, DEFAULT_HEIGHT);
47.       g.setColor(Color.RED);
48.       g.fillOval(DEFAULT_WIDTH / 4, DEFAULT_WIDTH / 4, DEFAULT_WIDTH / 2,
49.                  DEFAULT_HEIGHT / 2);
50.
```

```
51.       label.setIcon(new ImageIcon(image));
52.       add(new JScrollPane(label), BorderLayout.CENTER);
53.       JPanel panel = new JPanel();
54.
55.       JButton copyButton = new JButton("Copy");
56.       panel.add(copyButton);
57.       copyButton.addActionListener(new ActionListener()
58.          {
59.             public void actionPerformed(ActionEvent event)
60.                { copy(); }
61.          });
62.
63.       JButton pasteButton = new JButton("Paste");
64.       panel.add(pasteButton);
65.       pasteButton.addActionListener(new ActionListener()
66.          {
67.             public void actionPerformed(ActionEvent event)
68.                {
69.                 paste();
70.                }
71.          });
72.
73.       add(panel, BorderLayout.SOUTH);
74.    }
75.
76.    /**
77.     * Копирует текущее изображение в системный буфер обмена.
78.     */
79.    private void copy()
80.    {
81.       Clipboard clipboard =
82.          Toolkit.getDefaultToolkit().getSystemClipboard();
83.       ImageTransferable selection = new ImageTransferable(image);
84.       clipboard.setContents(selection, null);
85.    }
86.
87.    /**
88.     * Вставляет изображение из системного
89.     * буфера обмена в метку.
90.     */
91.    private void paste()
92.    {
93.       Clipboard clipboard =
94.          Toolkit.getDefaultToolkit().getSystemClipboard();
95.       DataFlavor flavor = DataFlavor.imageFlavor;
96.       if (clipboard.isDataFlavorAvailable(flavor))
97.       {
98.          try
99.          {
100.            image = (Image) clipboard.getData(flavor);
101.            label.setIcon(new ImageIcon(image));
102.         }
103.         catch (UnsupportedFlavorException exception)
104.         {
105.            JOptionPane.showMessageDialog(this, exception);
106.         }
```

```
107.        catch (IOException exception)
108.        {
109.           JOptionPane.showMessageDialog(this, exception);
110.        }
111.     }
112.  }
113.
114.  private JLabel label;
115.  private Image image;
116.
117.  private static final int DEFAULT_WIDTH = 300;
118.  private static final int DEFAULT_HEIGHT = 300;
119. }
120.
121. /**
122.  * Класс-оболочка для передачи объектов-изображений.
123.  */
124. class ImageTransferable implements Transferable
125. {
126.    /**
127.     * Выбор изображения.
128.     * @param image Изображение
129.     */
130.    public ImageTransferable(Image image)
131.    {
132.       theImage = image;
133.    }
134.
135.    public DataFlavor[] getTransferDataFlavors()
136.    {
137.       return new DataFlavor[] { DataFlavor.imageFlavor };
138.    }
139.
140.    public boolean isDataFlavorSupported(DataFlavor flavor)
141.    {
142.       return flavor.equals(DataFlavor.imageFlavor);
143.    }
144.
145.    public Object getTransferData(DataFlavor flavor)
146.       throws UnsupportedFlavorException
147.    {
148.       if(flavor.equals(DataFlavor.imageFlavor))
149.       {
150.          return theImage;
151.       }
152.       else
153.       {
154.          throw new UnsupportedFlavorException(flavor);
155.       }
156.    }
157.
158.    private Image theImage;
159. }
```

Передача объектов Java через системный буфер обмена

Предположим, что объекты требуется копировать и вставлять из одного Java-приложения в другое. В таком случае использовать локальный буфер обмена нельзя. Но зато, к счастью, можно использовать системный буфер путем помещения в него *сериализируемых* (seriliazed) объектов Java.

Эта возможность демонстрируется на примере программы, приведенной в листинге 7.13. Данная программа отображает палитру цветов для выбора. При выполнении щелчка на кнопке Copy (Копировать) она копирует выбранный в текущий момент цвет в системный буфер обмена в виде сериализируемого объекта `Color`, а при выполнении щелчка на кнопке Paste (Вставить) проверяет, содержится ли в системном буфере объект `Color`, и если да, извлекает его оттуда и делает текущим выбранным вариантом в палитре цветов.

Сериализируемые объекты могут передаваться и между двумя Java-приложениями (рис. 7.43). Чтобы убедиться в этом, запустите два экземпляра программы `SerialTransferTest`, затем щелкните на кнопке Copy в первом и на кнопке Paste во втором, и объект `Color` будет передан от одной виртуальной машины другой.

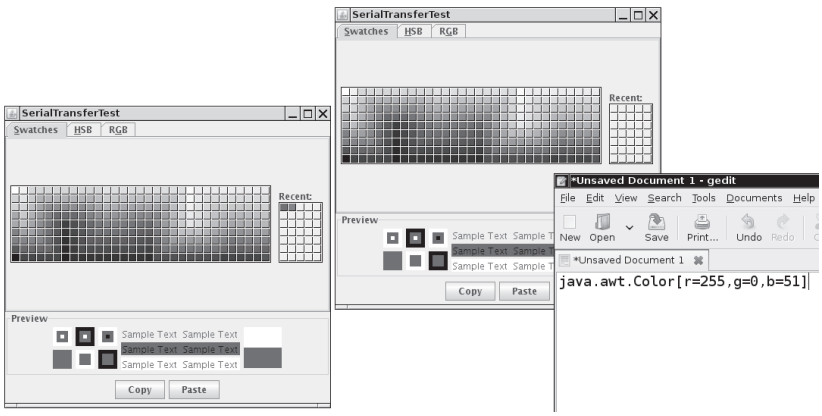


Рис. 7.43. Копирование данных между двумя экземплярами Java-приложения

Для обеспечения возможности передачи данных платформа Java предусматривает использование формата двоичных данных для системного буфера, содержащего сериализируемый объект, чтобы другие Java-программы (не обязательного такого же типа, как и та, что сгенерировала данные для этого буфера обмена) могли извлекать содержащиеся в буфере данные и десериализировать объект.

Конечно, приложения, созданные на языках, отличных от Java, не будут знать, что делать с такими данными. Поэтому в нашей демонстрационной программе для них предусматривается еще один формат, а именно — текстовый формат, который получается путем применения по отношению к передаваемому объекту метода `toString()`. Чтобы увидеть его, запустите программу, выберите какой-нибудь цвет и затем щелкните на кнопке Paste в текстовом редакторе. Это приведет к появлению в окне текстового редактора приблизительно следующей строки:

```
java.awt.Color[r=255,g=0,b=51]
```

По сути, писать никакого дополнительного кода для обеспечения передачи сериализируемых объектов не требуется. Достаточно просто использовать MIME-тип:

```
application/x-java-serialized-object;class=имя_класса
```

Класс-оболочку для передачи данных, как и раньше, нужно создавать самостоятельно. Как именно — можно увидеть в коде демонстрационной программы.

Листинг 7.13. Содержимое файла `SerialTransferTest.java`

```
1.  import java.io.*;
2.  import java.awt.*;
3.  import java.awt.datatransfer.*;
4.  import java.awt.event.*;
5.  import javax.swing.*;
6.
7.  /**
8.   * Эта программа демонстрирует осщуествление передачи
9.   * сериализируемых объектов между виртуальными машинами.
10.  * @version 1.02 2007-08-16
11.  * @author Cay Horstmann
12.  */
13. public class SerialTransferTest
14. {
15.   public static void main(String[] args)
16.   {
17.     EventQueue.invokeLater(new Runnable()
18.       {
19.         public void run()
20.         {
21.           JFrame frame = new SerialTransferFrame();
22.           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.           frame.setVisible(true);
24.         }
25.     });
26.   }
27. }
28.
29. /**
30.  * Фрейм с палитрой цветов и кнопками для их копирования и вставки.
31.  */
32. class SerialTransferFrame extends JFrame
33. {
34.   public SerialTransferFrame()
35.   {
36.     setTitle("SerialTransferTest");
37.
38.     chooser = new JColorChooser();
39.     add(chooser, BorderLayout.CENTER);
40.     JPanel panel = new JPanel();
41.
42.     JButton copyButton = new JButton("Copy");
43.     panel.add(copyButton);
44.     copyButton.addActionListener(new ActionListener()
45.       {
46.          public void actionPerformed(ActionEvent event)
47.         {
48.            copy();
49.         }
50.     });
51.
```

```
52.    JButton pasteButton = new JButton("Paste");
53.    panel.add(pasteButton);
54.    pasteButton.addActionListener(new ActionListener()
55.      {
56.        public void actionPerformed(ActionEvent event)
57.        {
58.          paste();
59.        }
60.      });
61.
62.    add(panel, BorderLayout.SOUTH);
63.    pack();
64.  }
65.
66.   /**
67.    * Копирование выбранного в палитре цвета в системный буфер обмена.
68.    */
69.    private void copy()
70.    {
71.     Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
72.      Color color = chooser.getColor();
73.      SerialTransferable selection = new SerialTransferable(color);
74.      clipboard.setContents(selection, null);
75.    }
76.   /**
77.    * Вставка цвета из системного буфера обмена в палитру.
78.    */
79.   private void paste()
80.   {
81.    Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
82.    try
83.    {
84.       DataFlavor flavor = new DataFlavor(
85.         "application/x-java-serialized-object;class=java.awt.Color");
86.       if (clipboard.isDataFlavorAvailable(flavor))
87.       {
88.         Color color = (Color) clipboard.getData(flavor);
89.         chooser.setColor(color);
90.       }
91.     }
92.     catch (ClassNotFoundException e)
93.     {
94.       JOptionPane.showMessageDialog(this, e);
95.     }
96.     catch (UnsupportedFlavorException e)
97.     {
98.        JOptionPane.showMessageDialog(this, e);
99.     }
100.    catch (IOException e)
101.    {
102.     JOptionPane.showMessageDialog(this, e);
103.     }
104.   }
105.
106.  private JColorChooser chooser;
107. }
```

```
108. /**
109.  * Класс-оболочка для осуществления передачи сериализируемых объектов.
110.  */
111. class SerialTransferable implements Transferable
112. {
113.   /**
114.    * Выбор объекта.
115.    * @param o Любой сериализируемый объект
116.    */
117.   SerialTransferable(Serializable o)
118.   {
119.     obj = o;
120.   }
121.
122.   public DataFlavor[] getTransferDataFlavors()
123.   {
124.     DataFlavor[] flavors = new DataFlavor[2];
125.     Class<?> type = obj.getClass();
126.     String mimeType = "application/x-java-serialized-object;class=" + type.getName();
127.
128.     try
129.     {
130.       flavors[0] = new DataFlavor(mimeType);
131.       flavors[1] = DataFlavor.stringFlavor;
132.       return flavors;
133.     }
134.     catch (ClassNotFoundException e)
135.     {
136.       return new DataFlavor[0];
137.     }
138.   }
139.
140.    public boolean isDataFlavorSupported(DataFlavor flavor)
141.    {
142.      return DataFlavor.stringFlavor.equals(flavor)
143.       || "application".equals(flavor.getPrimaryType())
144.       && "x-java-serialized-object".equals(flavor.getSubType())
145.       && flavor.getRepresentationClass().isAssignableFrom(obj.getClass());
146.     }
147.
148.     public Object getTransferData(DataFlavor flavor)
149.        throws UnsupportedFlavorException
150.     {
151.     if (!isDataFlavorSupported(flavor))
152.       throw new UnsupportedFlavorException(flavor);
153.
154.     if (DataFlavor.stringFlavor.equals(flavor)) return obj.toString();
155.     return obj;
156.  }
157.
158.   private Serializable obj;
159. }
```

Использование локального буфера обмена для передачи ссылок на объекты

Иногда может возникать необходимость в копировании и вставке данных такого типа, которые не поддерживаются системным буфером обмена, то есть не является сериализируемыми. Для передачи ссылки на произвольный объект Java в пределах одной и той же виртуальной машины (JVM) применяется тип MIME:

```
application/x-java-jvm-local-objectref;class=имя_класса
```

Для этого типа требуется определять объект-оболочку `Transferable`. Необходимый для этого процесс полностью аналогичен процессу определения объекта-оболочки `SerialTransferable`, который приводился в предыдущем разделе.

Ссылка на объект имеет значение только в пределах одной виртуальной машины. Поэтому копировать такой объект в системный буфер обмена нельзя. Вместо этого следует использовать локальный буфер обмена:

```
Clipboard clipboard = new Clipboard("local");
```

В качестве параметра конструктору нужно передавать желаемое имя для буфера обмена.

Однако у подхода с использованием локального буфера обмена имеется один серьезный недостаток. Он требует обеспечения синхронизации между локальным и системным буферами, чтобы пользователи их не путали. На момент написания этой книги автоматическое обеспечение такой синхронизации платформой Java еще не поддерживалось.

API **java.awt.datatransfer.Clipboard 1.1**

- `Clipboard(String name)`
  Создает локальный буфер обмена с указанным именем.

Копирование путем перетаскивания

При использовании механизма копирования и вставки для передачи информации между двумя программами в роли посредника всегда выступает буфер обмена. Механизм *перетаскивания* (drag and drop) исключает участие этого посредника и позволяет двум программам взаимодействовать напрямую. Платформа Java предусматривает базовую поддержку для этого механизма. В частности, она позволяет налаживать возможность осуществления операций перетаскивания между приложениями Java и платформенно-ориентированными программами. В этом разделе демонстрируется пример создания как Java-приложения для выполнения роли приемника перетаскиваемых объектов, так и приложения для выполнения роли источника перетаскиваемых объектов.

Прежде чем углубляться в изучение предлагаемой платформой Java поддержки для операций перетаскивания, давайте сначала вкратце рассмотрим пользовательский интерфейс перетаскивания на примере программ Windows Explorer и WordPad. На другой платформе вместо этих программ для экспериментирования можно взять и другие доступные программы с возможностями перетаскивания.

Инициируется *операция перетаскивания* путем выполнения соответствующего *жеста* внутри *источника перетаскивания*, то есть путем сначала выбора одного или более элементов и затем их перетаскивания из первоначального места в какое-нибудь другое. После отпускания кнопки мыши над целевым местом, то есть приемником пе-

ретаскивания, приемник запрашивает у источника информацию о перетаскиваемых элементах и выполняет соответствующую операцию. Например, в случае перетаскивания пиктограммы файла в окне диспетчера файлов поверх пиктограммы какого-нибудь каталога, произойдет перемещение этого файла в этот каталог. Однако в случае ее перетаскивания в окно текстового редактора, произойдет открытие этого файла в окне текстового редактора. (Это, конечно же, подразумевает использование такого диспетчера файлов и текстового редактора, которые поддерживают операции перетаскивания, как, например, программы Explorer и WordPad в Windows или Nautilus и `gedit` в Gnome.)

Нажатие и удержание клавиши <Ctrl> при перетаскивании приведет к замене операции *перемещения* операцией *копирования* и помещению в каталог лишь *копии* файла, а нажатие и удержание одновременно и клавиши <Ctrl> и клавиши <Shift> — вообще к созданию в каталоге ссылающегося на этот файл *ярлыка*. (На других платформах для выполнения этих операций могут быть предусмотрены другие сочетания клавиш.)

Таким образом, получается, что операции перетаскивания могут подразумевать выполнение трех различных действий:

- перемещения;
- копирования;
- создание ярлыка.

Под созданием ярлыка, по сути, подразумевается создание ссылки на перетаскиваемый элемент. Такие ссылки обычно требуют поддержки со стороны обслуживающей операционной системы (вроде символических ссылок для файлов или объектных ссылок для компонентов документов), и обычно не имеют особого смысла в программах, рассчитанных на использование на различных платформах. Поэтому в этом разделе основное внимание будет уделяться применению только тех операций перетаскивания, которые подразумевают копирование и перемещение объектов.

Обычно операции перетаскивания сопровождаются специальными визуальными эффектами. Самым простым из них является изменение формы курсора мыши. При наведении курсора мыши на потенциальный пункт назначения (приемник), его форма изменяется и начинает указывать на то, является ли выполнение операции перетаскивания в этот пункт назначения возможным. Если выполнение этой операции возможно, тогда форма курсора еще также указывает и на то, какое именно действие разрешается выполнить. В табл. 7.6 описано несколько различных форм, которые курсор может принимать при перетаскивании.

Таблица 7.6. Формы, которые курсор может принимать при перетаскивании

Действие	Значок в Windows	Значок в Gnome
Перемещение		
Копирование		
Создание ярлыка		
Выполнение операции перетаскивания невозможно		

Перетаскивать можно не только пиктограммы файлов, но и другие элементы. Например, можно выделять какой-то текст в редакторе WordPad или `gedit` и перетас-

кивать его. Попробуйте перетащить фрагменты текста в разные места и посмотрите, что будет происходить.



Подобный эксперимент продемонстрирует недостатки механизма перетаскивания как части пользовательского интерфейса. Он может вызывать у пользователей сомнения по поводу того, какие именно элементы они могут перетаскивать, куда именно они их могут перетаскивать, и к чему именно это будет приводить. Из-за того, что выполняемая по умолчанию при перетаскивании операция перемещения может приводить к удалению оригинала, многие пользователи, по вполне понятным причинам, стараются избегать экспериментировать с перетаскиванием.

Поддержка передачи данных в компонентах Swing

Начиная с Java SE 1.4, несколько компонентов Swing имеет встроенную поддержку для операций перетаскивания (табл. 7.7). Некоторые из этих компонентов позволяют выбирать текст для перетаскивания, а некоторые — соответственно, вставлять его. Для обеспечения обратной совместимости требуется вызывать метод `setDragEnabled`, активизирующий возможность перетаскивания. Возможность вставки (dropping) активна всегда.



В пакете `java.awt.dnd` поставляется низкоуровневый API-интерфейс перетаскивания, отвечающий за предоставление базовых возможностей перетаскивания в Swing. Этот API в настоящей книге не рассматривается.

Таблица 7.7. Поддержка передачи данных в компонентах Swing

Компонент	Источник перетаскивания	Приемник перетаскивания
`JFileChooser`	Экспортирует список файлов	—
`JColorChooser`	Экспортирует объект, представляющий цвет	Принимает объекты, представляющие цвета
`JTextField` `JFormatedTextField`	Экспортирует выбранный текст	Принимает текст
`JPasswordField`	Отсутствует (из соображений безопасности)	Принимает текст
`JTextArea` `JTextPane` `JEditorPane`	Экспортирует выбранный текст	Принимает текст и списки файлов
`JList` `JLabel` `JTree`	Экспортирует описание выбранного текста (то есть только его копию)	—

Программа, код которой приведен в листинге 7.14, демонстрирует поведение всех этих компонентов. После запуска этой программы следует обратить внимание на следующие моменты:

- на то, что из списка, таблицы и дерева можно выбирать и перетаскивать по нескольку элементов;
- на то, что перетаскивание элементов из таблицы осуществляется немного неудобным образом, то есть на то, что сначала нужно выбирать элемент с помо-

щью мыши, затем отпускать кнопку мыши, после чего снова выполнять щелчок и только затем выполнять перетаскивание;

- на то, что при перетаскивании элементов в текстовую область можно видеть, как происходит форматирование перетаскиваемой информации, то есть, как ячейки таблицы разделяются символами табуляции и каждая выбранная строка размещается в отдельной строке (рис. 7.44);
- на то, что 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 модели данных; о том, как такую возможность можно реализовать, когда модель данных является редактируемой, будет рассказываться в следующем разделе);
- на то, что перетаскивать элементы в список, таблицу, дерево или селектор файлов нельзя;
- на то, что в случае запуска двух копий программы представляющий цвет элемент можно перетаскивать из одного селектора цветов в другой;
- на то, что перетаскивать текст из текстовой области нельзя, поскольку для нее не вызывался метод `setDragEnabled`.

В пакете Swing предусмотрен потенциально полезный механизм для быстрого превращения компонента из источника перетаскивания в приемник. И заключается он в установке для соответствующего свойства обработчика передачи данных (transfer handler). Например, в нашей демонстрационной программе присутствует такой вызов:

```
textField.setTransferHandler(new TransferHandler("background"));
```

Благодаря нему, перетаскивание цвета в текстовое поле будет приводить к изменению цвета его фона.

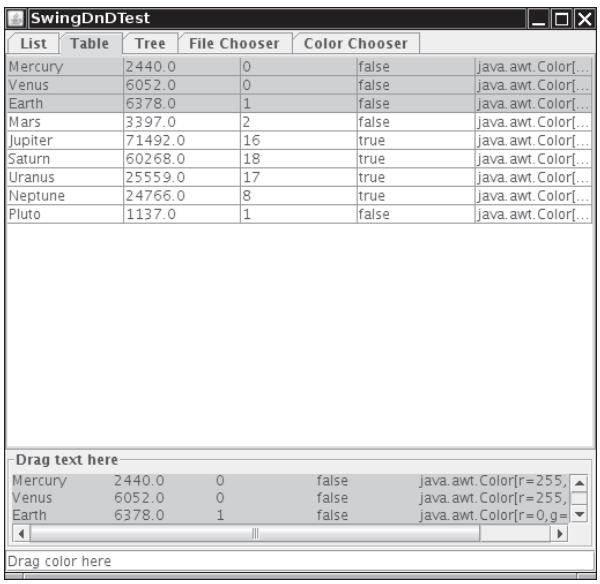


Рис. 7.44. Программа для тестирования поддерживаемых в Swing возможностей перетаскивания

При осуществлении перетаскивания этот обработчик передачи данных будет проверять, имеется ли у одного из типов данных класс представления `Color`, и если да, вызывать метод `setBackground`.

Установка этого обработчика в текстовом поле означает отключение стандартного обработчика, из-за чего в этом текстовом поле больше нельзя будет ни вырезать, ни вставлять, ни перетаскивать текст, но зато из него можно будет перетаскивать цвет. Для инициации действия перетаскивания какой-нибудь текст в нем все же выбрать придется. При перетаскивании этого текста будет обнаруживаться, что его можно вставлять в селектор цвета и изменять значение его цвета на цвет фона текстового поля. Однако перетаскивать текст в саму текстовую область будет невозможно.

Листинг 7.14. Содержимое файла `SwingDnDTest.java`

```
1.  import java.awt.*;
2.
3.  import javax.swing.*;
4.  import javax.swing.border.*;
5.  import javax.swing.event.*;
6.
7.  /**
8.   * Данная программа демонстрирует поддерживаемые в Swing
9.   * базовые возможности для осуществления перетаскивания
10.  * @version 1.10 2007-09-20
11.  * @author Cay Horstmann.
12.  */
13. public class SwingDnDTest
14. {
15.    public static void main(String[] args)
16.    {
17.      EventQueue.invokeLater(new Runnable()
18.       {
19.         public void run()
20.        {
21.         JFrame frame = new SwingDnDFrame();
22.         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.         frame.setVisible(true);
24.       }
25.    });
26. }
27. class SwingDnDFrame extends JFrame
28. {
29.    public SwingDnDFrame()
30.    {
31.       setTitle("SwingDnDTest");
32.       JTabbedPane tabbedPane = new JTabbedPane();
33.
34.       JList list = SampleComponents.list();
35.       tabbedPane.addTab("List", list);
36.       JTable table = SampleComponents.table();
37.       tabbedPane.addTab("Table", table);
38.       JTree tree = SampleComponents.tree();
39.       tabbedPane.addTab("Tree", tree);
40.       JFileChooser fileChooser = new JFileChooser();
41.       tabbedPane.addTab("File Chooser", fileChooser);
42.       JColorChooser colorChooser = new JColorChooser();
43.       tabbedPane.addTab("Color Chooser", colorChooser);
```

```
44.       final JTextArea textArea = new JTextArea(4, 40);
45.       JScrollPane scrollPane = new JScrollPane(textArea);
46.       scrollPane.setBorder(new TitledBorder(new EtchedBorder(), "Drag text  here"));
47.
48.       JTextField textField = new JTextField("Drag color here");
49.       textField.setTransferHandler(new TransferHandler("background"));
50.
51.      tabbedPane.addChangeListener(new ChangeListener()
52.       {
53.        public void stateChanged(ChangeEvent e)
54.        {
55.          textArea.setText("");
56.        }
57.     });
58.
59.     tree.setDragEnabled(true);
60.     table.setDragEnabled(true);
61.     list.setDragEnabled(true);
62.     fileChooser.setDragEnabled(true);
63.     colorChooser.setDragEnabled(true);
64.     textField.setDragEnabled(true);
65.
66.     add(tabbedPane, BorderLayout.NORTH);
67.     add(scrollPane, BorderLayout.CENTER);
68.     add(textField, BorderLayout.SOUTH);
69.     pack();
70.   }
71. }
```

API javax.swing.JComponent 1.2

- `void setTransferHandler(TransferHandler handler)` 1.4

  Устанавливает обработчик передачи данных для управления операциями передачи данных (то есть операциями вырезания, копирования, вставки и перетаскивания).

API javax.swing.TransferHandler 1.4

- `TransferHandler(String propertyName)`

  Создает обработчик передачи данных, предусматривающий осуществление считывания или записи свойства компонента JavaBeans с указанным именем при выполнении той или иной операции по передаче данных.

API javx.swing.JFileChooser 1.2

API javax.swing.JColorChooser 1.2

API javax.swing.JTextComponent 1.2

API javax.swing.JList 1.2

API javax.swing.JTable 1.2

API javax.swing.JTree 1.2

- `void setDragEnabled(boolean b)` 1.4

  Включает или отключает возможность перетаскивания данных из соответствующего компонента.

Источники перетаскивания

В предыдущем разделе было показано, как можно пользоваться базовыми возможностями для перетаскивания, которые предлагаются в компонентах Swing. В этом же разделе демонстрируется, как конфигурировать любой компонент так, чтобы он выступал в роли источника перетаскивания (drag source), а в следующем — как превращать компоненты в приемники перетаскивания (drag target), а также пример реализации компонента, выступающего в роли как источника, так и приемника изображений.

Настраивать касающееся перетаскивания поведение компонента Swing можно путем создания нового подкласса `TransferHandler` и затем переопределения в нем сначала метода `getSourceActions` так, чтобы он указывал, какие именно действия (копирование, перемещение, создание ярлыка) должен поддерживать компонент, а потом — метода `getTransferable`, чтобы он выдавал объект `Transferable`, посредством того же самого процесса, который применяется для осуществления копирования данных в буфер обмена.

В нашей демонстрационной программе предполагается, что требуется обеспечить возможность перетаскивания изображений из заполненного пиктограммами изображений компонента `JList` (рис. 7.45).

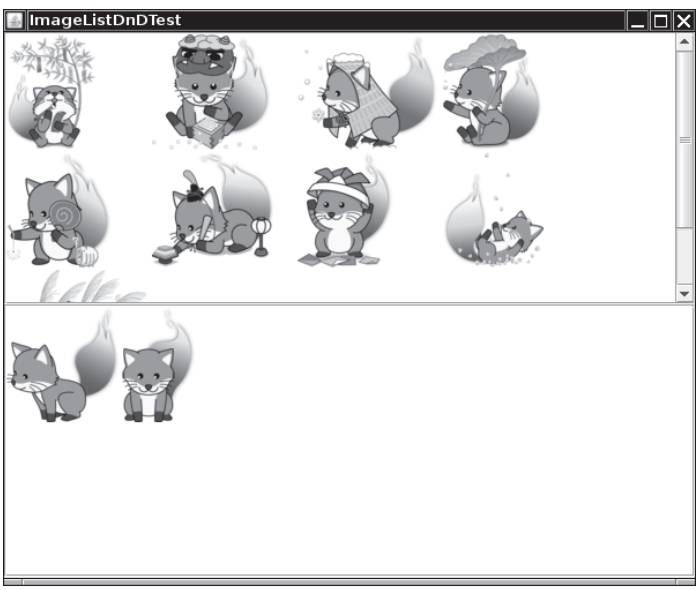


Рис. 7.45. Приложение с поддерживающим возможность перетаскивания списком изображений

Поэтому реализация метода createTransferable в ней выглядит так, как показано ниже, а объект выбираемого изображения просто помещается в оболочку 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());
}
```

В нашем примере компонент JList уже, к счастью, предусматривает возможность распознавания жеста перетаскивания и инициации соответствующего процесса передачи данных. Остается только активизировать этот механизм путем вызова метода setDragEnabled. В случае добавления поддержки перетаскивания для компонента, который не способен распознавать такой жест, однако, процесс передачи данных придется инициировать самостоятельно. Например, ниже показано, как его можно будет инициировать в случае компонента 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);
  }
});
```

Этот код предусматривает инициацию процесса передачи данных просто при выполнении пользователем щелчка на данной надписи. Более сложная реализация могла бы предусматривать наблюдение за таким движением мыши, при котором мышь перетаскивается на небольшое расстояние.

При завершении пользователем операции перетаскивания вызывается такой метод обработчика передачи данных источника, как exportDone. В случае если пользователь выполнил операцию перемещения, в этом методе должно осуществляться удаление переданного объекта. Следовательно, для списка изображений код реализации этого метода должен выглядеть так:

```
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);
  }
}
```

В общем, получается, что для превращения компонента в источник перетаскивания требуется просто добавлять обработчик передачи данных, указывающий на следующее:

- какие действия должны поддерживаться;
- какие данные должны передаваться;
- каким образом исходные данные должны удаляться после операции перемещения.

Помимо этого, в случае превращения в источник перетаскивания компонента, который не входит в список компонентов, перечисленных в табл. 7.7, еще также дополнительно потребуется обеспечивать наблюдение за передвижениями (жестами) мыши и инициацию процесса передачи данных.

API javax.swing.TransferHandler 1.4

- `int getSourceActions(JComponent c)`

  Должен переопределяться так, чтобы он возвращал (в виде поразрядной операции или комбинации, состоящей из `COPY`, `MOVE` и `LINK`) информацию о тех действиях, которые источнику разрешено выполнять при перетаскивании данных из указанного компонента.

- `protected Transferable createTransferable(JComponent source)`

  Должен переопределяться так, чтобы он создавал для подлежащих перетаскиванию данных объект `Transferable`.

- `void exportAsDrag(JComponent comp, InputEvent e, int action)`

  Запускает процесс перетаскивания из указанного компонента. Для параметра `action` может указываться одно из следующих значений: `COPY`, `MOVE` или `LINK`.

- `protected void exportDone(JComponent source, Transferable data, int action)`

  Должен переопределяться так, чтобы он вынуждал источник перетаскивания после успешного завершения передачи данных выполнять необходимое действие.

Приемники перетаскивания

В этом разделе демонстрируется, как можно реализовать приемник перетаскивания (drag target). В качестве примера, опять-таки, используется компонент `Jlist` с пиктограммами изображений. Предполагается, что необходимо обеспечить для пользователей возможность перетаскивания изображений *в* список.

Превращение компонента в приемник перетаскивания осуществляется путем установки `TransferHandler` и реализации методов `canImport` и `importData`.



В случае использования версии Java SE 6 обработчик передачи данных можно добавить к компоненту `JFrame`. Именно этот компонент наиболее часто применяется для отпускания (dropping) файлов в приложении после перетаскивания. К числу допустимых мест отпускания относятся декорации фрейма и строка меню, но не содержащиеся внутри фрейма компоненты (у которых имеются свои собственные обработчики передачи данных).

Метод `canImport` вызывается постоянно, когда пользователь наводит курсор мыши на компонент, выступающий в роли приемника перетаскивания. Если тот разрешает отпускание, метод возвращает `true`. Именно эта информации и влияет на внешний вид пиктограммы курсора, дающий визуальную подсказку о том, является выполнение операции отпускания возможным или нет.

Начиная с версии Java SE 6, метод `canImport` имеет параметр типа `TransferHandler.TransferSupport`. Через этот параметр можно получать информацию о действии, которое пользователь выбрал для осуществления отпускания, о месте выполнения отпускания и, наконец, о данных, которые требуется передать после отпускания. (До Java SE 6 существовала другая версия метода `canImport`, которая позволяла получать только перечень разновидностей данных.)

В методе `canImport` можно еще также и перекрывать выбранное пользователем действие для отпускания. Например, если пользователь выбрал в качестве действия перемещение, но оно не подходит для удаления оригинала, можно вынудить обработчик передачи данных использовать вместо него копирование.

Ниже приведен типичный пример. Компонент списка изображений готов принимать отпускаемые списки файлов и изображений. Однако если список файлов перетаскивается в этот компонент, выбранное пользователем действие `MOVE` заменяется действием `COPY` для того, чтобы не происходило удаление файлов изображений.

```
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 еще выполняться и проверка на предмет того, действительно ли в файлах содержатся изображения. Такие компоненты Swing, как `JList`, `JTable`, `JTree` и `JTextComponent`, предусматривают отображение визуальной подсказки о месте отпускания при наведении курсора мыши на поверхность приемника перетаскивания. По умолчанию для обозначения места отпускания применяется либо выделение (если речь идет о компонентах вроде `JList`, `JTable` и `JTree`), либо символ вставки (если речь идет о компоненте `TextComponent`). Такой подход не является ни удобным для пользователя, ни гибким, и применяется по умолчанию исключительно в целях обеспечения обратной совместимости. Для выбора более подходящей визуальной подсказки следует вызывать метод `setDropMode`.

Помимо этого еще также можно управлять тем, должны ли отпускаемые данные перезаписывать существующие элементы либо же вставляться между ними. Например, в нашей демонстрационной программе для предоставления пользователю возможности осуществлять отпускание прямо поверх элемента (то есть заменять его) или вставлять данные между двумя элементами (рис. 7.46) выполняется следующий вызов:

```
setDropMode(DropMode.ON_OR_INSERT);
```

В табл. 7.8 перечислены все режимы отпускания, которые поддерживаются компонентами Swing.

Таблица 7.8. Режимы отпускания

Компонент	Поддерживаемые режимы отпускания
`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 символ вставки, а не выделенный фрагмент)



Рис. 7.46. Визуальные подсказки о выполнении отпускания поверх другого элемента или вставки между двумя элементами

После завершения пользователем жеста отпускания сразу же вызывается метод `importData`, требующий получения данных из источника перетаскивания. Вызов метода `getTransferable` на параметре `TransferSupport` позволяет получать ссылку на объект `Transferable`, который предоставляет тот же самый интерфейс, что применяется для копирования и вставки.

Одним из наиболее часто используемых для перетаскивания и отпускания типов данных является `DataFlavor.javaFileListFlavor`. С помощью списка файлов описывается набор перетаскиваемых в приемник файлов. Передаваемые данные помещаются в объект типа `List<File>`. Код, необходимый для извлечения этих файлов, выглядит так:

```
DataFlavor[] flavors = transferable.getTransferDataFlavors();
if (Arrays.asList(flavors).contains(DataFlavor.javaFileListFlavor))
{
  List<File> fileList =
     (List<File>) transferable.getTransferData(DataFlavor.javaFileListFlavor);
  for (File f : fileList)
  {
    Выполнение каких-то действий с f;
  }
}
```

При выполнении перетаскивания в какой-нибудь из тех компонентов, которые перечислены в табл. 7.8, требуется в точности знать место отпускания данных. Для выяснения этого места можно вызывать метод `getDropLocation` на параметре `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 с помощью целочисленного индекса, а вот в дереве — с помощью древовидного пути. Код, необходимый для выяснения места отпускания в нашем списке изображений, выглядит так:

```
int index;
if (support.isDrop())
{
  JList.DropLocation location = (JList.DropLocation) support.getDropLocation();
  index = location.getIndex();
}
else index = model.size();
```

У подкласса `JList.DropLocation` имеется метод `getIndex`, который возвращает индекс места отпускания. (У подкласса `JTree.DropLocation` для выполнения той же функции предусмотрен метод `getPath`.)

Метод `importData` еще также вызывается и при вставке данных в компонент путем нажатия комбинации клавиш <Ctrl+V>. В таком случае метод `getDropLocation` будет выдавать исключение `IllegalStateException`. Поэтому, чтобы этого не происходило, если метод `isDrop` возвращает `false`, вставляемые данные просто добавляются в конец списка.

При выполнении вставки данных в список, таблицу или дерево, еще также обязательно должна осуществляться проверка на предмет того, должны ли эти данные быть вставлены между элементами или заменить тот элемент, который находится в месте их отпускания. В случае списка такую проверку можно обеспечить путем вызова метода `isInsert` подкласса `JList.DropLocation`. Что касается остальных компонентов, то информацию о необходимых для них методах и подклассах можно найти во врезках API в конце этой главы.

В общем, получается, что для превращения компонента в приемник перетаскивания требуется просто добавлять обработчик передачи данных, указывающий на следующее:

- когда перетаскиваемый элемент может приниматься;
- каким образом должны импортироваться отпускаемые данные.

В листинге 7.15 приведен полный код демонстрационной программы. Обратите внимание на то, что класс `ImageList` выполняет роль как источника, так и приемника перетаскивания. Попробуйте перетащить изображения между двумя списками. Можете также попробовать перетащить в эти списки файлы изображений из селектора файлов какой-то другой программы.

Листинг 7.15. Содержимое файла `ImageListDragDrop.java`

```
1.  import java.awt.*;
2.  import java.awt.datatransfer.*;
3.  import java.io.*;
4.  import java.util.*;
5.  import javax.imageio.*;
6.  import javax.swing.*;
7.  import java.util.List;
8.
9.  /**
10.  * Эта программа демонстрирует обеспечениме поддержки перетаскивания
11.  * и отпускания элементов в списке изображений.
12.  * @version 1.00 2007-09-20
13.  * @author Cay Horstmann
14.  */
15. public class ImageListDnDTest
16. {
17.   public static void main(String[] args)
18.   {
19.     EventQueue.invokeLater(new Runnable()
20.       {
21.         public void run()
22.         {
23.           JFrame frame = new ImageListDnDFrame();
24.           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25.           frame.setVisible(true);
26.         }
27.       });
28.     }
29.   }
30.
31. class ImageListDnDFrame extends JFrame
32. {
33.   public ImageListDnDFrame()
34.   {
```

```
35.     setTitle("ImageListDnDTest");
36.     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
37.
38.     list1 = new ImageList(new File("images1").listFiles());
39.     list2 = new ImageList(new File("images2").listFiles());
40.     setLayout(new GridLayout(2, 1));
41.     add(new JScrollPane(list1));
42.     add(new JScrollPane(list2));
43.   }
44.
45.   private ImageList list1;
46.   private ImageList list2;
47.   private static final int DEFAULT_WIDTH = 600;
48.   private static final int DEFAULT_HEIGHT = 500;
49. }
50.
51. class ImageList extends JList
52. {
53.   public ImageList(File[] imageFiles)
54.   {
55.     DefaultListModel model = new DefaultListModel();
56.     for (File f : imageFiles)
57.       model.addElement(new ImageIcon(f.getPath()));
58.
59.     setModel(model);
60.     setVisibleRowCount(0);
61.     setLayoutOrientation(JList.HORIZONTAL_WRAP);
62.     setDragEnabled(true);
63.     setDropMode(DropMode.ON_OR_INSERT);
64.     setTransferHandler(new ImageListTransferHandler());
65.   }
66. }
67.
68. class ImageListTransferHandler extends TransferHandler
69. {
70.   // Поддержка для перетаскивания.
71.
72.   public int getSourceActions(JComponent source)
73.   {
74.     return COPY_OR_MOVE;
75.   }
76.
77.   protected Transferable createTransferable(JComponent source)
78.   {
79.     JList list = (JList) source;
80.     int index = list.getSelectedIndex();
81.     if (index < 0) return null;
82.     ImageIcon icon = (ImageIcon) list.getModel().getElementAt(index);
83.     return new ImageTransferable(icon.getImage());
84.   }
85.
86.   protected void exportDone(JComponent source, Transferable data, int action)
87.   {
88.     if (action == MOVE)
89.     {
90.       JList list = (JList) source;
91.       int index = list.getSelectedIndex();
```

```
92.       if (index < 0) return;
93.       DefaultListModel model = (DefaultListModel) list.getModel();
94.       model.remove(index);
95.     }
96.   }
97.
98.   // Поддержка для отпускания.
99.
100.  public boolean canImport(TransferSupport support)
101.  {
102.    if (support.isDataFlavorSupported(DataFlavor.javaFileListFlavor))
103.    {
104.      if (support.getUserDropAction() == MOVE) support.setDropAction(COPY);
105.      return true;
106.    }
107.    else return support.isDataFlavorSupported(DataFlavor.imageFlavor);
108.  }
109.
110.  public boolean importData(TransferSupport support)
111.  {
112.    JList list = (JList) support.getComponent();
113.    DefaultListModel model = (DefaultListModel) list.getModel();
114.
115.    Transferable transferable = support.getTransferable();
116.    List<DataFlavor> flavors = Arrays.asList(transferable.getTransferDataFlavors());
117.    List<Image> images = new ArrayList<Image>();
118.
119.    try
120.    {
121.      if (flavors.contains(DataFlavor.javaFileListFlavor))
122.      {
123.        List<File> fileList = (List<File>) transferable
124.            .getTransferData(DataFlavor.javaFileListFlavor);
125.        for (File f : fileList)
126.        {
127.          try
128.          {
129.            images.add(ImageIO.read(f));
130.          }
131.          catch (IOException ex)
132.          {
133.            // не удалось считать изображение--пропустить
134.          }
135.        }
136.      }
137.      else if (flavors.contains(DataFlavor.imageFlavor))
138.      {
139.        images.add((Image) transferable
140.            .getTransferData(DataFlavor.imageFlavor));
141.      }
142.      int index;
143.      if (support.isDrop())
144.      {
145.        JList.DropLocation location = (JList.DropLocation) support.getDropLocation();
146.        index = location.getIndex();
147.        if (!location.isInsert()) model.remove(index); // замена
148.      }
```

```
149.      else index = model.size();
150.      for (Image image : images)
151.      {
152.        model.add(index, new ImageIcon(image));
153.        index++;
154.      }
155.        return true;
156.     }
157.     catch (IOException ex)
158.     {
159.      return false;
160.     }
161.    catch (UnsupportedFlavorException ex)
162.    {
163.      return false;
164.    }
165.   }
166. }
```

API javax.swing.TransferHandler 1.4

- `boolean canImport(TransferSupport support) 6`

  Должен переопределяться так, чтобы он указывал, может ли целевой компонент принимать перетаскиваемый объект, описанный в параметре `TransferSupport`.

- `boolean importData(TransferSupport support) 6`

  Должен переопределяться так, чтобы он выполнял только описанную в параметре `TransferSupport` операцию отпускания или вставки и возвращал `true`, если операция импорта прошла успешно.

API javax.swing.JFrame 1.2

- `void setTransferHandler(TransferHandler handler) 6`

  Устанавливает обработчик передачи данных для обработки только операций отпускания и вставки.

API javax.swing.JList 1.2

API javax.swing.JTable 1.2

API javax.swing.JTree 1.2

API javax.swing.JTextComponent 1.2

- `void setDropMode(DropMode mode) 6`

  Устанавливает для режима отпускания данного компонента одно из перечисленных в табл. 7.8 значений.

API javax.swing.TransferHandler.TransferSupport 6

- `Component getComponent()`

  Получает информацию о целевом компоненте в данной операции передачи данных.

- `DataFlavor[] getDataFlavors()`

  Получает информацию о типах подлежащих передаче данных.

- `boolean isDrop()`

  Возвращает `true`, если данная операция передачи данных подразумевает отпускание данных, и `false`, если она подразумевает их вставку.

- `int getUserDropAction()`

  Получает информацию о действии, выбранном пользователем для осуществления отпускания (действием может быть `MOVE`, `COPY` или `LINK`).

- `getSourceDropActions()`

  Получает информацию о тех действиях отпускания, которые разрешает выполнять источник перетаскивания.

- `getDropAction()`
- `setDropAction()`

  Получает или задает действие отпускания для данной операции передачи данных. Первоначально это действие выбирается пользователем, но далее оно может и перекрываться обработчиком передачи данных.

- `DropLocation getDropLocation()`

  Получает информацию о месте отпускания или выдает исключение `IllegalStateException`, если операция передачи данных подразумевает выполнение не отпускания, а какого-то другого действия (например, вставки).

API javax.swing.TransferHandler.DropLocation 6

- `Point getDropPoint()`

  Получает информацию о месте положения курсора мыши при выполнении отпускания в целевом компоненте (компоненте-приемнике).

API javax.swing.JList.DropLocation 6

- `boolean isInsert()`

  Возвращает `true`, если данные должны быть вставлены перед указанным местом, и `false`, если они должны заменить собой уже существующие данные.

- `int getIndex()`

  Получает индекс модели для вставки или замены.

API javax.swing.JTable.DropLocation 6

- `boolean isInsertRow()`
- `boolean isInsertColumn()`

  Возвращает `true`, если данные должны быть вставлены перед строкой или столбцом.

- `int getRow()`
- `int getColumn()`

  Получает индекс модели или столбца для выполнения вставки или замены или `-1`, если отпускание произошло в пустой области.

API javax.swing.JTree.DropLocation 6

- `TreePath getPath()`
- `int getChildIndex()`

  Возвращает путь в дереве и индекс того дочернего элемента, которые вместе с режимом отпускания целевого компонента определяют место отпускания, как описано в табл. 7.9.

Таблица 7.9. Определение места отпускания в компоненте `JTree`

Режим отпускания	Операция редактирования, которая должна выполняться в дереве
`INSERT`	Вставлять в виде дочернего элемента пути, перед дочерним элементом с данным индексом
`ON` или `USE_SELECTION`	Заменять данные пути (индекс дочернего элемента не используется)
`INSERT_OR_ON`	Если индекс дочернего элемента выглядит как `-1`, поступать как в режиме `ON`, в противном случае — как в режиме `INSERT`.

API javax.swing.JTextComponent.DropLocation 6

- `int getIndex()`

  Возвращает позицию индекса, в которой должны быть вставлены данные.

Средства интеграции в платформу

И в завершении этой главы давайте рассмотрим несколько средств, которые были добавлены в Java SE 6 для придания Java-приложениям большего родства с платформой. Средство создания начальных экранов-заставок позволяет делать так, чтобы при запуске виртуальной машины Java-приложение сразу же отображало начальный экран. Класс `java.awt.Desktop` позволяет обеспечивать в Java-приложениях возможность запуска собственных приложений платформы, вроде используемого по умолчанию браузера или программы электронной почты. И, наконец, еще одно средство позволяет получать из Java-приложений доступ к системному лотку, также называемому областью пиктограмм панели задач (system tray), и заполнять его необходимыми значками подобно тому, как это делают многие собственные приложения платформы.

Начальные экраны-заставки

Наиболее распространенной жалобой по поводу Java-приложений является то, что они слишком долго запускаются. У виртуальной машины Java всегда уходит какое-то время на загрузку всех требуемых классов, особенно в случае приложений Swing, для которых ей приходится загружать довольно большие объемы кода Swing и AWT. Пользователи не любят приложения, при запуске которых до отображения первого экрана проходит много времени, и могут даже пытаться запускать подобные приложения по нескольку раз, когда не знают, удалась ли их первая попытка. Удобным ре-

шением для этой проблемы является *начальный экран-заставка* (splash screen), то есть небольшое окно, быстро появляющееся и информирующее пользователя о том, что приложение было успешно запущено.

Оснащение Java-приложений подобными экранами традиционно считается трудной задачей. Конечно, сразу же после запуска метода `main` можно отображать окно. Но ведь запуск метода `main` происходит только после того, как загрузчик классов загрузит все зависимые классы, а это может занимать некоторое время.

В Java SE 6 эта проблема решается путем предоставления виртуальной машине возможности отображать любое необходимое изображение сразу же при запуске. Для указания этого изображения доступно два механизма. Первый заключается в использовании в командной строке опции `-splash`:

```
java -splash:myimage.png MyApp
```

Второй механизм предусматривает указание требуемого изображения в манифесте внутри файла JAR:

```
Main-Class: MyApp
SplashScreen-Image: myimage.gif
```

Это изображение будет отображаться сразу при запуске, и автоматически исчезать при появлении первого окна AWT. Использовать разрешается любые изображения GIF, JPEG и PNG. Эффекты анимации (в GIF) и прозрачности (GIF и PNG) тоже поддерживаются.

Тем, чье приложение готово приступать к работе сразу же, как только достигает метода `main`, могут не читать остальную часть этого раздела. Однако многие приложения все-таки имеют модульную архитектуру, в которой небольшое ядро загружает при запуске ряд подключаемых модулей. Eclipse и NetBeans являются типичными тому примерами. В таком случае на начальном экране-заставке может отображаться ход процесса загрузки таких подключаемых модулей.

Для получения такого эффекта существует два подхода. Первый подразумевает выполнение рисования прямо на начальном экране-заставке, а второй — замену этого экрана не имеющим рамок фреймом с идентичным содержимым и выполнение рисования внутри уже этого фрейма. В нашей демонстрационной программе иллюстрируются оба этих подхода.

Для выполнения рисования прямо на экране-заставке сначала получается ссылка на экран-заставку и информация о его графическом контексте и размерах:

```
SplashScreen splash = SplashScreen.getSplashScreen();
Graphics2D g2 = splash.createGraphics();
Rectangle bounds = splash.getBounds();
```

После этого может выполняться рисование самым обычным образом. По завершении должен обязательно вызываться метод `update` для гарантированного обновления рисунка. В нашей демонстрационной программе осуществляется рисование простой панели индикации хода выполнения, которую можно увидеть в левой части рис. 7.47.

```
g.fillRect(x, y, width * percent / 100, height);
splash.update();
```



Начальный экран-заставка является одноэлементным объектом. Создавать самостоятельно подобный собственный объект нельзя. В случае не указания экрана-заставки ни в командной строке, ни манифесте, метод `getSplashScreen` будет возвращать `null`.



Рис. 7.47. Первоначальный экран-заставка и последующее окно без рамок

У подхода с выполнением рисования прямо на экране-заставке есть один недостаток, и заключается он в утомительности вычисления всех пиксельных позиций и несовпадении прорисовываемой панели индикации хода выполнения с родной. Во избежание этих проблем может быть лучше заменять при запуске метода `main` начальный экран-заставку последующим окном с тем же размером и содержимым. Такое окно может содержать любые компоненты Swing.

Наша демонстрационная программа, код которой приведен в листинге 7.16, показывает применение и такого подхода тоже. В правой части рис. 7.47 можно видеть не имеющий рамок фрейм с панелью, в котором прорисовывается экран-заставка и содержится компонент `JProgressBar`. Добавление такого фрейма открывает полный доступ к API-интерфейсу Swing и позволяет легко добавлять строки с сообщениями, не беспокоясь ни о каких пиксельных позициях.

Обратите внимание на то, что удалять первоначальный экран-заставку не требуется. Он будет исчезать автоматически, как только будет становиться видимым последующее окно.



К сожалению, при замене экрана-заставки последующим окном будет происходить заметное мерцание.

Листинг 7.16. Содержимое файла `SplashScreenTest.java`

```
1.  import java.awt.*;
2.  import java.util.List;
3.  import javax.swing.*;
4.
5.   /**
6.    * Эта программа демонстрирует API начального экрана-заставки.
7.    * @version 1.00 2007-09-21
8.    * @author Cay Horstmann
9.    */
```

```
10. public class SplashScreenTest
11. {
12.   private static void drawOnSplash(int percent)
13.   {
14.     Rectangle bounds = splash.getBounds();
15.     Graphics2D g = splash.createGraphics();
16.     int height = 20;
17.     int x = 2;
18.     int y = bounds.height - height - 2;
19.     int width = bounds.width - 4;
20.     Color brightPurple = new Color(76, 36, 121);
21.     g.setColor(brightPurple);
22.     g.fillRect(x, y, width * percent / 100, height);
23.     splash.update();
24.   }
25.
26.   /**
27.    * Этот метод выполянет рисование прямо на экране-заставке.
28.    */
29.    private static void init1()
30.    {
31.      splash = SplashScreen.getSplashScreen();
32.      if (splash == null)
33.      {
34.        System.err.println("Did you specify a splash image
35.                           with -splash or in the manifest?");
36.                       // Указано ли изображение для заставки
37.                       // с помощью опции –splash или манифеста?
38.        System.exit(1);
39.      }
40.
41.     try
42.     {
43.       for (int i = 0; i <= 100; i++)
44.       {
45.         drawOnSplash(i);
46.         Thread.sleep(100); // Имитация работы при запуске.
47.       }
48.      }
49.      catch (InterruptedException e)
50.      {
51.      }
52.    }
53.
54.   /**
55.    * Этот метод отображает фрейм с тем же
56.    * изображением, что и на экране-заставке.
57.    */
58.    private static void init2()
59.    {
60.       final Image img = Toolkit.getDefaultToolkit().getImage(splash.getImageURL());
61.
62.      final JFrame splashFrame = new JFrame();
63.      splashFrame.setUndecorated(true);
64.
```

```
65.      final JPanel splashPanel = new JPanel()
66.       {
67.         public void paintComponent(Graphics g)
68.         {
69.            g.drawImage(img, 0, 0, null);
70.         }
71.      };
72.
73.    final JProgressBar progressBar = new JProgressBar();
74.    progressBar.setStringPainted(true);
75.    splashPanel.setLayout(new BorderLayout());
76.    splashPanel.add(progressBar, BorderLayout.SOUTH);
77.
78.    splashFrame.add(splashPanel);
79.    splashFrame.setBounds(splash.getBounds());
80.    splashFrame.setVisible(true);
81.
82.    new SwingWorker<Void, Integer>()
83.    {
84.      protected Void doInBackground() throws Exception
85.      {
86.        try
87.        {
88.         for (int i = 0; i <= 100; i++)
89.         {
90.           publish(i);
91.           Thread.sleep(100);
92.         }
93.      }
94.      catch (InterruptedException e)
95.      {
96.      }
97.      return null;
98.    }
99.
100.   protected void process(List<Integer> chunks)
101.   {
102.     for (Integer chunk : chunks)
103.     {
104.       progressBar.setString("Loading module " + chunk); // (Загрузка модуля)
105.       progressBar.setValue(chunk);
106.       splashPanel.repaint();   // потому что изображения
107.                                // загружаются асинхронным образом
108.     }
109.   }
110.
111.   protected void done()
112.   {
113.     splashFrame.setVisible(false);
114.     JFrame frame = new JFrame();
115.     frame.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
116.     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
117.     frame.setTitle("SplashScreenTest");
118.     frame.setVisible(true);
119.   }
120.  }.execute();
121. }
```

```
122.  public static void main(String args[])
123.  {
124.     init1();
125.
126.     EventQueue.invokeLater(new Runnable()
127.      {
128.        public void run()
129.        {
130.           init2();
131.        }
132.     });
133.   }
134.
135.   private static SplashScreen splash;
136.   private static final int DEFAULT_WIDTH = 300;
137.   private static final int DEFAULT_HEIGHT = 300;
138. }
```

API java.awt.SplashScreen 6

- `static SplashScreen getSplashScreen()`
  Получает ссылку на экран-заставку или `null`, если таковой отсутствует.
- `URL getImageURL()`
- `void setImageURL(URL imageURL)`
  Получает или устанавливает URL-адрес изображения для экрана-заставки. Установка изображения приводит к обновлению экрана-заставки.
- `Rectangle getBounds()`
  Получает информацию о границах экрана-заставки.
- `Graphics2D createGraphics()`
  Получает графический контекст для выполнения рисования на экране-заставке.
- `void update()`
  Обновляет отображаемый экран-заставку.
- `void close()`
  Закрывает экран-заставку. Когда становится видимым первое окно AWT, экран-заставка закрывается автоматически.

Запуск настольных приложений

Класс `java.awt.Desktop` позволяет запускать такие используемые по умолчанию в настольной системе приложения, как браузер и программа электронной почты, а также открывать, редактировать и распечатывать файлы с помощью тех приложений, которые зарегистрированы в системе как подлежащие использованию для работы с файлами такого типа.

API-интерфейс выглядит очень просто. Сначала вызывается статический метод `isDesktopSupported`. Если он возвращает `true`, это означает, что текущая платформа поддерживает возможность запуска настольных приложений и тогда вызывается статический метод `getDesktop` для получения экземпляра `Desktop`.

Не все настольные среды поддерживают все операции API. Например, в настольной среде Gnome в Linux операция открытия файлов поддерживается, а операция их распечатки нет (потому что в ней не предусмотрена поддержка использования “глаго-

лов" в ассоциациях файлов). Выяснение того, какие операции поддерживаются на данной платформе, а какие нет, осуществляется путем вызова метода `isSupported` и передачи ему в качестве аргумента одного из значений перечисления `Desktop.Action`. В нашей демонстрационной программе содержатся проверки, подобные той, что показана ниже:

```
if (desktop.isSupported(Desktop.Action.PRINT)) printButton.setEnabled(true);
```

То есть для открытия, редактирования или распечатки файла сначала выполняется проверка на предмет того, поддерживается ли вообще такая операция, а затем уже вызывается метод `open`, `edit` или `print`. Для запуска браузера передается `URI`. (Об URI-адресах более подробно рассказывалось в главе 3.) Можно и просто вызывать конструктор `URI` со строкой, содержащей URL-адрес типа `http` или `https`.

Для запуска используемой по умолчанию программы электронной почты необходимо создавать `URI` определенного формата, а именно такого, который показан ниже:

```
mailto:recipient?query
```

Здесь на месте *`recipient`* должен содержаться электронный адрес получателя, наподобие `president@whitehouse.gov`, а на месте *`query`* — разделенные символом `&` пары "имя-значение" с закодированными с помощью знака `%` значениями, вроде `subject=dinner%20RSVP&bcc=putin%40kremvax.ru`. (Алгоритм кодирования с помощью знака `%` выглядит практически так же, как и описанный в главе 3 алгоритм кодирования с помощью URL, но только пробел представляется не как `+`, а как `%20`.) Этот формат документирован в RFC 2368 (`http://www.ietf.org/rfc/rfc2368.txt`).

К сожалению, классу `URI` об URL-адресах типа `mailto` ничего не известно, поэтому компоновать и кодировать их нужно самостоятельно. Даже еще хуже то, что на момент написания этой книги еще не было принято никакого стандарта для обработки не ASCII символов. Наиболее распространенным подходом (которым пользуемся и мы) является преобразование каждого такого символа в UTF-8 и кодирование получающихся байтов с использованием знака `%`.

Наша демонстрационная программа, код которой приведен в листинге 7.17, позволяет открывать, редактировать или распечатывать любой выбираемый файл, отображать в окне браузера любой желаемый URL-адрес, а также запускать свою программу для работы с электронной почтой (рис. 7.48).

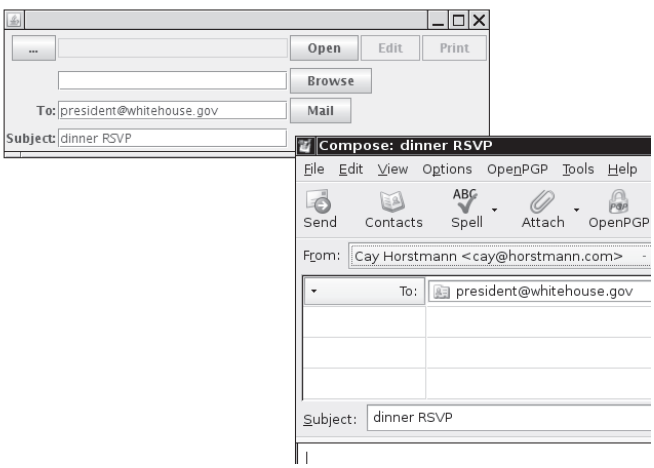


Рис. 7.48. Запуск настольного приложения

Листинг 7.17. Содержимое файла `DesktopAppTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.io.*;
4.  import java.net.*;
5.  import javax.swing.*;
6.  /**
7.   * Эта программа демонстрирует API настольных приложений.
8.   * @version 1.00 2007-09-22
9.   * @author Cay Horstmann
10.  */
11. public class DesktopAppTest
12. {
13.  public static void main(String[] args)
14.  {
15.  EventQueue.invokeLater(new Runnable()
16.    {
17.       public void run()
18.       {
19.         JFrame frame = new DesktopAppFrame();
20.         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.         frame.setVisible(true);
22.        }
23.     });
24.   }
25. }
26. class DesktopAppFrame extends JFrame
27. {
28.  public DesktopAppFrame()
29.  {
30.    setLayout(new GridBagLayout());
31.    final JFileChooser chooser = new JFileChooser();
32.    JButton fileChooserButton = new JButton("...");
33.    final JTextField fileField = new JTextField(20);
34.    fileField.setEditable(false);
35.    JButton openButton = new JButton("Open");
36.    JButton editButton = new JButton("Edit");
37.    JButton printButton = new JButton("Print");
38.    final JTextField browseField = new JTextField();
39.    JButton browseButton = new JButton("Browse");
40.    final JTextField toField = new JTextField();
41.    final JTextField subjectField = new JTextField();
42.    JButton mailButton = new JButton("Mail");
43.    openButton.setEnabled(false);
44.    editButton.setEnabled(false);
45.    printButton.setEnabled(false);
46.    browseButton.setEnabled(false);
47.    mailButton.setEnabled(false);
48.    if (Desktop.isDesktopSupported())
49.    {
50.      Desktop desktop = Desktop.getDesktop();
51.      if (desktop.isSupported(Desktop.Action.OPEN)) openButton.setEnabled(true);
52.      if (desktop.isSupported(Desktop.Action.EDIT)) editButton.setEnabled(true);
53.      if (desktop.isSupported(Desktop.Action.PRINT)) printButton.setEnabled(true);
54.      if (desktop.isSupported(Desktop.Action.BROWSE)) browseButton.setEnabled(true);
55.      if (desktop.isSupported(Desktop.Action.MAIL)) mailButton.setEnabled(true);
56.    }
```

```
57.     fileChooserButton.addActionListener(new ActionListener()
58.      {
59.        public void actionPerformed(ActionEvent e)
60.        {
61.          if (chooser.showOpenDialog(DesktopAppFrame.this) ==
62.            JFileChooser.APPROVE_OPTION)
63.          fileField.setText(chooser.getSelectedFile().getAbsolutePath());
64.        }
65.      });
66.     openButton.addActionListener(new ActionListener()
67.       {
68.         public void actionPerformed(ActionEvent e)
69.         {
70.           try
71.           {
72.             Desktop.getDesktop().open(chooser.getSelectedFile());
73.         }
74.         catch (IOException ex)
75.         {
76.           ex.printStackTrace();
77.         }
78.       }
79.     });
80.   editButton.addActionListener(new ActionListener()
81.     {
82.      public void actionPerformed(ActionEvent e)
83.      {
84.        try
85.         {
86.           Desktop.getDesktop().edit(chooser.getSelectedFile());
87.         }
88.         catch (IOException ex)
89.         {
90.           ex.printStackTrace();
91.         }
92.       }
93.    });
94.   printButton.addActionListener(new ActionListener()
95.    {
96.       public void actionPerformed(ActionEvent e)
97.       {
98.        try
99.        {
100.          Desktop.getDesktop().print(chooser.getSelectedFile());
101.       }
102.      catch (IOException ex)
103.      {
104.        ex.printStackTrace();
105.       }
106.    }
107.  });
108.  browseButton.addActionListener(new ActionListener()
109.     {
110.      public void actionPerformed(ActionEvent e)
111.     {
112.       try
113.       {
114.         Desktop.getDesktop().browse(new URI(browseField.getText()));
115.       }
```

```
116.       catch (URISyntaxException ex)
117.       {
118.          ex.printStackTrace();
119.       }
120.       catch (IOException ex)
121.      {
122.        ex.printStackTrace();
123.      }
124.    }
125.  });
126.  mailButton.addActionListener(new ActionListener()
127.   {
128.     public void actionPerformed(ActionEvent e)
129.     {
130.       try
131.      {
132.        String subject = percentEncode(subjectField.getText());
133.        URI uri = new URI("mailto:" + toField.getText() + "?subject=" + subject);
134.        System.out.println(uri);
135.        Desktop.getDesktop().mail(uri);
136.      }
137.      catch (URISyntaxException ex)
138.      {
139.         ex.printStackTrace();
140.      }
141.      catch (IOException ex)
142.      {
143.         ex.printStackTrace();
144.      }
145.     }
146.   });
147.   JPanel buttonPanel = new JPanel();
148.   ((FlowLayout) buttonPanel.getLayout()).setHgap(2);
149.    buttonPanel.add(openButton);
150.    buttonPanel.add(editButton);
151.    buttonPanel.add(printButton);
152.    add(fileChooserButton, new GBC(0, 0).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).setInsets(5, 2, 5, 2));
158.    add(toField, new GBC(1, 2).setFill(GBC.HORIZONTAL));
159.    add(mailButton, new GBC(2, 2).setAnchor(GBC.WEST).setInsets(2));
160.    add(new JLabel("Subject:"), new GBC(0, 3).setAnchor(GBC.EAST).setInsets(5, 2, 5, 2));
161.    add(subjectField, new GBC(1, 3).setFill(GBC.HORIZONTAL));
162.    pack();
163.  }
164.  private static String percentEncode(String s)
165.  {
166.    try
167.    {
168.       return URLEncoder.encode(s, "UTF-8").replaceAll("[+]", "%20");
169.    }
170.    catch (UnsupportedEncodingException ex)
171.    {
172.      return null; // UTF-8 всегда поддерживается
173.     }
174.   }
175. }
```

API java.awt.Desktop 6

- `static boolean isDesktopSupported()`

  Возвращает `true`, если данная платформа поддерживает возможность запуска настольных приложений.

- `static Desktop getDesktop()`

  Возвращает объект `Desktop` для запуска настольных операций. Выдает исключение `UnsupportedOperationException`, если данная платформа не поддерживает возможность запуска настольных операций.

- `boolean isSupported(Desktop.Action action)`

  Возвращает `true`, если указанное действие поддерживается. А указываться 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-адресом.

- `void mail()`
- `void mail(URI uri)`

  Запускает используемую по умолчанию программу для работы с почтой. Вторая версия может применяться для заполнения соответствующих частей электронного сообщения.

Системный лоток

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 *системным лотком* (system tray), а отображаемые в ней пиктограммы, соответственно — *пиктограммами лотка* (tray icons). В API Java используется точно такая же терминология. Типичным примером подобной программы является программа-монитор, которая следит за появлением обновленных версий программного обеспечения. При появлении обновлений она может изменять внешний вид отображаемой в системном лотке пиктограммы или отображать рядом с ней соответствующее сообщение.

Честно говоря, системный лоток обычно и так насыщен пиктограммами, и добавление в него еще одной пиктограммы, как правило, не вызывает у пользователей особой радости. И наша демонстрационная программа (отображающая виртуальные "печенья с сюрпризом") вряд ли станет тому исключением.

Класс `java.awt.SystemTray` является универсальным (независящим от платформы) проводником к системному лотку. Так же как и в случае рассмотренного в предыдущем разделе класса `Desktop`, сначала вызывается статический метод `isSupported` для выполнения проверки на предмет того, поддерживает ли локальная платформа Java системный лоток. Если да, то тогда далее получается одноэлементный объект `SystemTray` путем вызова статического метода `getSystemTray`.

Наиболее важным методом класса SystemTray является метод add, который позволяет добавлять экземпляр TrayIcon. У любой пиктограммы лотка имеется три следующих ключевых свойства:

- рисунок, который должен отображаться на этой пиктограмме;
- всплывающая подсказка, которая должна становиться видимой при наведении на эту пиктограмму курсора мыши;
- всплывающее меню, которое должно появляться при выполнении пользователем на этой пиктограмме щелчка правой кнопкой мыши.

Всплывающее меню добавляется с помощью экземпляра такого класса в библиотеке AWT, как PopupMenu, и представляет собой платформенно-ориентированное всплывающее меню, а не меню Swing, а необходимые пункты в него добавляются с помощью экземпляров, такого AWT-класса, как MenuItem, у каждого из которых, как и у Swing-аналогов, имеется свой слушатель действий.

И, наконец, пиктограмма лотка еще также может отображать пользователю различные уведомления (рис. 7.49). Обеспечивается подобное поведение путем вызова метода displayMessage класса TrayIcon и указания заголовка, текста и типа сообщения:

```
trayIcon.displayMessage("Your Fortune",
  fortunes.get(index),
  TrayIcon.MessageType.INFO);
```



Рис. 7.49. Уведомление от пиктограммы в системном лотке

В листинге 7.18 показано приложение, которое помещает в системный лоток пиктограмму “печенья с сюрпризом”. Это приложение считывает файл “печений с сюрпризом” (из известной UNIX-программы fortune), в котором каждое предсказание заканчивается строкой, содержащей символ %, и отображает новое сообщение каждые десять секунд. К счастью, у отображаемой им в системном лотке пиктограммы имеется всплывающее меню с пунктом, позволяющим завершить его работу. Вот если бы все отображаемые в системном лотке пиктограммы были настолько продуманными!

Листинг 7.18. Содержимое файла SystemTrayTest.java

```
1.  import java.awt.*;
2.  import java.util.*;
3.  import java.util.List;
4.  import java.awt.event.*;
5.  import java.io.*;
6.  import javax.swing.Timer;
7.
8.   /**
9.    * Эта программа демонстрирует использование API-интерфейса системного лотка.
10.   * @version 1.00 2007-09-22
11.   * @author Cay Horstmann
12.   */
13. public class SystemTrayTest
14. {
```

```
15.   public static void main(String[] args)
16.   {
17.     final TrayIcon trayIcon;
18.
19.     if (!SystemTray.isSupported())
20.     {
21.       System.err.println("System tray is not supported.");
22.                     // Системный лоток не поддерживается.
23.       return;
24.     }
25.
26.     SystemTray tray = SystemTray.getSystemTray();
27.     Image image = Toolkit.getDefaultToolkit().getImage("cookie.png");
28.
29.     PopupMenu popup = new PopupMenu();
30.     MenuItem exitItem = new MenuItem("Exit");
31.     exitItem.addActionListener(new ActionListener()
32.       {
33.         public void actionPerformed(ActionEvent e)
34.         {
35.           System.exit(0);
36.         }
37.     });
38.    popup.add(exitItem);
39.
40.    trayIcon = new TrayIcon(image, "Your Fortune", popup);
41.
42.    trayIcon.setImageAutoSize(true);
43.    trayIcon.addActionListener(new ActionListener()
44.     {
45.       public void actionPerformed(ActionEvent e)
46.       {
47.         trayIcon.displayMessage("How do I turn this off?",
48.           "Right-click on the fortune cookie and select Exit.",
49.           // Как отключить? Щелкните правой кнопкой мыши на "печенье
50.           // с сюрпризом" и выберите команду Exit (Выход).
51.            TrayIcon.MessageType.INFO);
52.       }
53.     });
54.
55.   try
56.   {
57.     tray.add(trayIcon);
58.   }
59.   catch (AWTException e)
60.   {
61.     System.err.println("TrayIcon could not be added.");
62.                   // Не удалось добавить пиктограмму в системный лоток.
63.     return;
64.   }
65.
66.   final List<String> fortunes = readFortunes();
67.   Timer timer = new Timer(10000, new ActionListener()
68.    {
69.      public void actionPerformed(ActionEvent e)
70.     {
71.       int index = (int) (fortunes.size() * Math.random());
```

```
72.       trayIcon.displayMessage("Your Fortune", fortunes.get(index),
73.         TrayIcon.MessageType.INFO);
74.     }
75.    });
76.   timer.start();
77.  }
78.
79.  private static List<String> readFortunes()
80.  {
81.  List<String> fortunes = new ArrayList<String>();
82.  try
83.   {
84.     Scanner in = new Scanner(new File("fortunes"));
85.     StringBuilder fortune = new StringBuilder();
86.     while (in.hasNextLine())
87.     {
88.       String line = in.nextLine();
89.       if (line.equals("%"))
90.       {
91.         fortunes.add(fortune.toString());
92.         fortune = new StringBuilder();
93.       }
94.      else
95.      {
96.        fortune.append(line);
97.         fortune.append(' ');
98.      }
99.    }
100.  }
101.  catch (IOException ex)
102.  {
103.    ex.printStackTrace();
104.  }
105.  return fortunes;
106.  }
107.
108. }
```

API java.awt.SystemTray 6

- `static boolean isSupported()`

  Возвращает `true`, если возможность доступа к системному лотку поддерживается на данной платформе.

- `static SystemTray getSystemTray()`

  Возвращает объект `SystemTray` для получения доступа к системному лотку. Выдает исключение `UnsupportedOperationException`, если данная платформа не поддерживает возможность доступа к системному лотку.

- `Dimension getTrayIconSize()`

  Получает размеры для пиктограммы с системном лотке.

- `void add(TrayIcon trayIcon)`
- `void remove(TrayIcon trayIcon)`

  Добавляет или удаляет пиктограмму системного лотка.

API java.awt.TrayIcon 6

- `TrayIcon(Image image)`
- `TrayIcon(Image image, String tooltip)`
- `TrayIcon(Image image, String tooltip, PopupMenu popupMenu)`

  Создает пиктограмму для системного лотка 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)`

  Получает или устанавливает изображение, всплывающую подсказку или всплывающее меню для данной всплывающей подсказки.

- `boolean isImageAutoSize()`

  Получает или устанавливает свойство `imageAutoSize`. Если устанавливает, тогда изображение масштабируется так, чтобы оно умещалось в области пиктограммы всплывающей подсказки, а если нет (что является поведением по умолчанию), изображение усекается (если оно является слишком большим) или размещается по центру (если оно является слишком маленьким).

- `void displayMessage(String caption, String text, TrayIcon.MessageType messageType)`

  Отображает сообщение рядом с пиктограммой лотка. Это может быть сообщение одного из следующих типов: `INFO`, `WARNING`, `ERROR` или `NONE`.

- `public void addActionListener(ActionListener listener)`
- `public void removeActionListener(ActionListener listener)`

  Добавляет или удаляет слушателя действий, когда вызываемый слушатель зависит от платформы. Обычно применяется при выполнении щелчка на уведомлении или выполнении двойного щелчка на пиктограмме в лотке.

Вот и подошла к концу эта длинная глава, посвященная расширенным средствам AWT. В следующей главе речь пойдет о спецификации JavaBeans и ее применении для сред разработки графических пользовательских интерфейсов.

Глава 8

КОМПОНЕНТЫ JAVABEANS

В ЭТОЙ ГЛАВЕ...

- Зачем нужны компоненты JavaBeans
- Создание компонентов JavaBeans
- Применение компонентов JavaBeans для создания приложений
- Соглашения об именовании свойств и событий компонента JavaBeans
- Типы свойств компонентов JavaBeans
- Классы `BeanInfo`
- Редакторы свойств
- Настройщики
- Постоянные компоненты JavaBeans

Официальное определение компонента bean в спецификации JavaBeans выглядит так: "Компонент bean — это многократно используемый элемент программного обеспечения, основанный на спецификации JavaBeans компании Sun, с которым можно работать в визуальном режиме с помощью инструментов интегрированной разработки".

После реализации bean-компонента одним разработчиком, другие разработчики могут сразу же начинать использовать его в среде интегрированной разработки (наподобие NetBeans). Вместо того чтобы писать утомительный код, они могут просто перетаскивать созданный другим разработчиком bean-компонент на форму GUI-интерфейса и настраивать его там нужным им образом с помощью диалоговых окон.

В этой главе рассказывается о том, как можно реализовывать bean-компоненты так, чтобы другим разработчикам было легко ими пользоваться.



Прежде чем продолжить изложение материала в этой главе, нам, во избежание возникновения всякого рода недоразумений, хотелось четко прояснить следующий момент: 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 для транзакций, постоянства, репликации и безопасности. Формально с ними тоже можно работать в средах разработки. Однако технология Enterprise JavaBeans является более сложной, чем предлагаемая в Standard Edition технология JavaBeans.

Вышесказанное вовсе не означает, что применение стандартных компонентов JavaBeans ограничивается лишь клиентскими программами. Web-технологии вроде JavaServer Faces (JSF) и JavaServer Pages (JSP) очень во многом полагаются на модель компонентов JavaBeans.

Зачем нужны компоненты JavaBeans

Программисты, имеющие опыт работы с Visual Basic, сразу понимают, почему bean-компоненты так важны. Программистам, привыкшим работать в средах, где все нужно делать самому, однако, часто оказывается трудно поверить в то, что Visual Basic является одним из наиболее успешных примеров технологии повторного использования объектов. Для тех, кому никогда не доводилось иметь дела с Visual Basic, ниже вкратце перечислены основные этапы создания простого приложения на этом языке.

1. Создание пользовательского интерфейса путем перетаскивания с помощью мыши компонентов (называемых в Visual Basic *элементами управления*) в окно формы.
2. Установка с помощью *инспекторов свойств* (property inspector) значений для свойств этих компонентов, например, для высоты, цвета или поведения.
3. Перечисление с помощью, опять-таки, инспекторов свойств событий, на которые эти компоненты могут реагировать либо путем использования соответствующих диалоговых окон, либо путем написания соответствующих коротких фрагментов обрабатывающего события кода.

Например, в главе 2 первого тома мы писали программу для отображения рисунков в фрейме. Ее код занял тогда у нас больше страницы. Чтобы создать программу с практически точно такими же функциональными возможностями в Visual Basic потребовалось бы выполнить всего лишь следующие действия.

1. Перетащить в окно фрейма два элемента управления, а именно — элемент управления `Image` для отображения графических рисунков и элемент `CommonDialog` для отображения окна, позволяющего выбирать файл.
2. Установить свойства `Filter` элемента управления `CommonDialog` так, чтобы отображались только те файлы, которые элемент управления `Image` может обрабатывать, как показано на рис. 8.1.

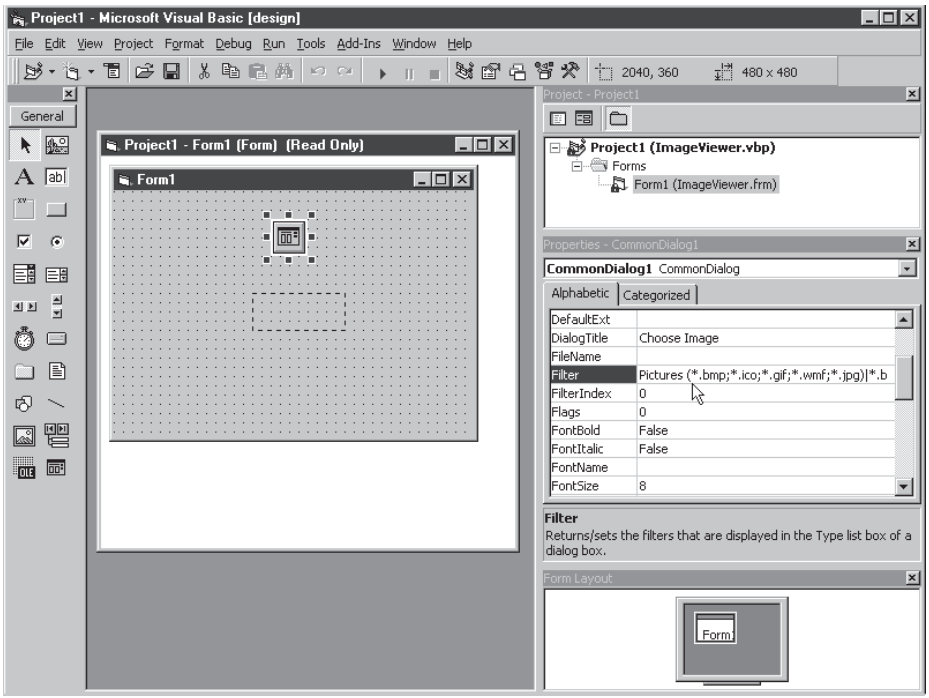


Рис. 8.1. Диалоговое окно Properties *(Свойства) в Visual Basic для приложения, отображающего рисунки*

3. Написать четыре следующих строки кода на языке Visual Basic, подлежащих активизации при первом запуске данного проекта:

```
Private Sub Form_Load()
  CommonDialog1.ShowOpen
  Image1.Picture = LoadPicture(CommonDialog1.FileName)
End Sub
```

Этот код отображает на экране диалоговое окно выбора файла, причем только с теми файлами, которые имеют правильное расширение, благодаря установленному соответствующим образом свойству `Filter`. После того как пользователь выберет нужный ему файл с рисунком, код дальше указывает элементу управления `Image` отобразить его.

Вот и все! Размещение элементов управления `CommonDialog` и `Image` и добавление соответствующих операторов позволяет добиться почти такого же эффекта, как и целая страница Java-кода. Конечно же, научиться перетаскивать готовые компоненты и задавать их свойства намного легче, чем научиться писать целые страницы кода.

Мы вовсе не хотим этим сказать, что Visual Basic является универсальным решением для любых задач. Вполне очевидно, что он является оптимальным решением лишь для задач конкретного рода, а именно — для создания Windows-приложений с развитым пользовательским интерфейсом. Технология JavaBeans была изобретена как раз для того, чтобы позволить технологии Java конкурировать с Visual Basic в этой самой области. Она предоставляет производителям программного обеспечения возможность создавать среды разработки в стиле Visual Basic, позволяющие создавать пользовательские интерфейсы с написанием минимального количества программного кода.

Создание компонентов JavaBeans

Написание компонентов JavaBeans не является сложным процессом с технической точки зрения, поскольку требует изучения всего лишь нескольких новых классов и интерфейсов. В частности, в простейшей форме компонент JavaBeans представляет собой не более чем, просто класс Java, следующий нескольким довольно строгим соглашениям в отношении именования его методов.



Некоторые специалисты утверждают, что компонент bean должен содержать конструктор по умолчанию. Авторы спецификации JavaBeans обходят этот вопрос молчанием. Следует отметить, что многие инструменты, входящие в состав интегрированных сред разработки, требуют для каждого компонента конструктор по умолчанию, чтобы иметь возможность создавать экземпляры соответствующих классов, не задавая параметров.

В листинге 8.1 содержится код компонента для просмотра изображений `ImageViewerBean`. Этот компонент JavaBeans можно использовать в любой интегрированной среде разработки Java. Он выполняет те же функции, что и элемент управления изображениями, созданный на Visual Basic, который упоминался в предыдущем разделе. Если внимательно изучить этот код, можно заметить, что класс `ImageViewerBean` практически ничем не отличается от других классов. Например, все методы для доступа к полям класса начинаются с префикса `get`, а все методы, изменяющие значения переменных, — с префикса `set`. Как показано ниже, для работы со *свойствами* в инструментах интегрированной среды разработки применяются стандартные соглашения об именах. Например, таким свойством компонента JavaBeans является `fileName`, поскольку оно имеет методы `get/set`.

В нашем примере свойство `fileName` образовано на основе переменной экземпляра `file`. Учтите: свойство отличается от переменной экземпляра тем, что свойства находятся на концептуально более высоком уровне, чем переменные экземпляра. Свойства являются атрибутами интерфейса, а переменные экземпляра относятся к реализации этого класса.

При рассмотрении приведенных в данной главе примеров следует иметь в виду, что реальные компоненты JavaBeans гораздо сложнее по двум причинам.

1. Компоненты JavaBeans должны быть доступны даже для не очень опытных специалистов. Им нужно предоставить доступ к функциональным возможностям компонента JavaBeans посредством визуальных средств разработки без дополнительного программирования.
2. Компоненты JavaBeans должны быть рассчитаны на *широкую область применения*. В них должны быть предусмотрены средства для настройки как поведения, так и внешнего вида. Это предполагает наличие большого количества свойств.

Хорошим примером компонента JavaBeans с богатым набором параметров является `CalendarBean` (рис. 8.2), созданный Каем Тодтером (Kai Toedter). Этот компонент и его исходный код можно бесплатно скопировать, обратившись по адресу: `http://www.toedter.com/en/jcalendar.html`. Компонент JavaBeans предоставляет пользователю удобный способ ввода даты за счет ее выбора на изображении календаря. Он имеет очень сложный код, который вряд ли следует создавать самостоятельно. Работая с технологией JavaBeans, можно воспользоваться готовыми результатами работы других программистов, перетаскивая компоненты в интегрированной среде разработки.

К счастью, для создания компонента JavaBeans с богатым набором свойств достаточно освоить лишь несколько основных принципов. Приведенные в данной главе примеры компонентов JavaBeans, хотя и не тривиальны, но слишком просты и подходят лишь для иллюстрации этих принципов.

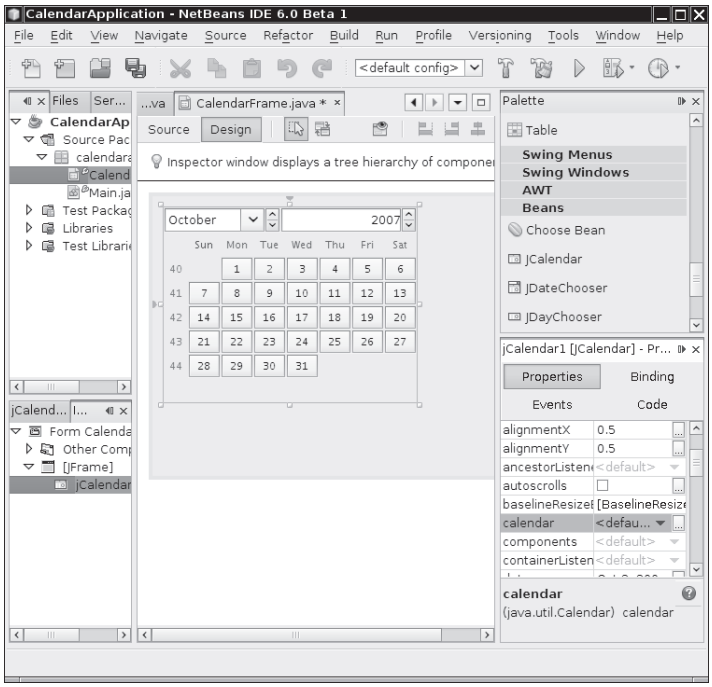


Рис. 8.2. Компонент, поддерживающий календарь

Листинг 8.1. Содержимое файла `ImageVieverBean.java`

```
1.  package com.horstmann.corejava;
2.
3.  import java.awt.*;
4.  import java.io.*;
5.  import javax.imageio.*;
6.  import javax.swing.*;
7.
8.  /**
9.    *Компонент bean, предназначенный для просмотра изображений.
10.   *@version 1.21 2001-08-15
11.   *@author Cay Horstmann
12. */
```

```
13. public class ImageViewerBean extends JLabel
14. {
15.
16.    public ImageViewerBean()
17.    {
18.       setBorder(BorderFactory.createEtchedBorder());
19.    }
20.
21.    /**
22.     * Установка свойства fileName.
23.     * @param fileName Имя файла, содержащего изображение
24.     */
25.    public void setFileName(String fileName)
26.    {
27.       try
28.       {
29.          file = new File(fileName);
30.          setIcon(new ImageIcon(ImageIO.read(file)));
31.       }
32.       catch (IOException e)
33.       {
34.          file = null;
35.          setIcon(null);
36.       }
37.    }
38.
39.    /**
40.     * Получение свойства fileName.
41.     * @return Имя файла, содержащего изображение
42.     */
43.    public String getFileName()
44.    {
45.       if (file == null) return null;
46.       else return file.getPath();
47.    }
48.
49.    public Dimension getPreferredSize()
50.    {
51.       return new Dimension(XPREFSIZE, YPREFSIZE);
52.    }
53.
54.    private File file = null;
55.    private static final int XPREFSIZE = 200;
56.    private static final int YPREFSIZE = 200;
57. }
```

Применение компонентов JavaBeans для создания приложений

Прежде чем углубляться в технические детали создания компонентов JavaBeans, давайте посмотрим, как эти компоненты можно применять и тестировать. Компонент `ImageViewerBean`, например, является вполне пригодным для использования компонентом, однако за пределами интегрированной среды разработки он утрачивает все свои особые возможности.

В каждой интегрированной среде разработки существует свой набор стратегий по упрощению жизни программиста. Здесь и далее рассматривается только одна среда разработки — NetBeans, доступная для загрузки по адресу `http://netbeans.org`.

В предлагаемом в этом разделе примере будет демонстрироваться использование двух компонентов JavaBeans — `ImageViewerBean` и `FileNameBean`. Код компонента `ImageViewerBean` уже приводился и объяснялся ранее, а код компонента `FileNameBean` будет анализироваться чуть позже в этой главе. Пока что все, что требуется знать — это то, что выполнение щелчка на кнопке с изображением троеточия (...) приводит к открытию окна, позволяющего выбрать файл.

Упаковка компонентов JavaBeans в JAR-файлах

Дабы сделать компонент JavaBeans пригодным для использования в среде разработки, все относящиеся к нему файлы классов необходимо упаковать в JAR-архиве. В отличие от JAR-архива, предназначенного для аплета, JAR-архив, предназначенный для компонента JavaBeans, нуждается в *файле манифеста* (manifest file), который описывает, какие файлы классов в архиве представляют компоненты JavaBeans и должны включаться в панель инструментов интегрированной среды разработки. Например, ниже показано содержимое файла манифеста `ImageViewerBean.mf` для компонента `ImageViewerBean`:

```
Manifest-Version: 1.0

Name: com/horstmann/corejava/ImageViewerBean.class
Java-Bean: True
```

Обратите внимание на наличие пустой строки между описанием версии манифеста и названием компонента JavaBeans.



Компоненты, рассматриваемые в качестве примера, мы помещаем в пакет `com.horstmann.corejava`, поскольку в некоторых средах разработки возникают проблемы при загрузке bean-компонентов из пакета по умолчанию.

Если компонент JavaBeans содержит несколько файлов классов, то в файле описания необходимо указать только файлы классов для тех компонентов JavaBeans, которые требуется отобразить в панели инструментов. Например, можно поместить компоненты `ImageViewerBean` и `FileNameBean` в один JAR-файл и использовать следующий файл описания:

```
 Manifest-Version: 1.0

Name: com/horstmann/corejava/ImageViewerBean.class
Java-Bean: True

Name: com/horstmann/corejava/FileNameBean.class
Java-Bean: True
```



Некоторые интегрированные среды разработки предъявляют чрезвычайно высокие требования к формату манифеста, например, в концах строк недопустимы пробелы, после определения версии и между записями компонентов JavaBeans должна располагаться пустая строка, а вслед за последней строкой должна располагаться еще одна пустая строка.

Для создания JAR-файла выполните перечисленные ниже действия.

1. Отредактируйте файл манифеста.
2. Соберите все необходимые файлы классов в одном каталоге.
3. Запустите архиватор `jar`, используя следующий синтаксис:

```
jar ctm jar_файл файл_манифеста файлы_классов
```

Пример команды, используемой для создания архива, приведен ниже:

```
jar ctm ImageViewerBean.jar ImageViewerBean.mf *.class
```

В JAR-файлы можно также включать другие объекты, например, пиктограммы в формате GIF. Пиктограммы в составе компонентов более подробно описываются далее в главе.



Убедитесь в том, что в JAR-архив упакованы все необходимые файлы, включая файлы внутренних классов, например, `FileNameBean$1.class`.

Интегрированная среда разработки обычно содержит механизмы включения новых компонентов JavaBeans, как правило, на основе загрузки JAR-архивов. Ниже перечислены те действия, которые нужно выполнить для включения новых компонентов в интегрированную среду разработки NetBeans.

Скомпилируйте классы `ImageViewerBean` и `FileNameBean` и упакуйте их в JAR-архив. Затем запустите интегрированную среду разработки NetBeans и выполните перечисленные ниже действия.

1. Выберите пункт меню Tools⇨Palette Manager (Сервис⇨Диспетчер палитр).
2. Щелкните на кнопке Add from JAR (Добавить из JAR-архива).
3. В диалоговом окне перейдите в каталог `ImageViewerBean` и выберите файл `ImageViewerBean.jar`.
4. После этого в диалоговом окне будет представлен список всех компонентов JavaBeans из выбранного JAR-архива. Выберите компонент `ImageViewerBean`.
5. Далее будет предложено указать, в какую палитру следует поместить компоненты. Выберите панель Beans. (Существуют и другие панели, например, для компонентов Swing, для компонентов AWT и т.д.)
6. Обратите внимание на палитру Beans. Теперь в ней содержится пиктограмма, которая представляет новый компонент JavaBeans. Однако в данном случае применяется пиктограмма по умолчанию. Далее в главе рассматриваются способы включения пиктограммы в компонент JavaBeans.

Повторите эти действия для включения компонента `FileNameBean`. Теперь на основе компонентов JavaBeans можно создать приложение.

Создание компонентов JavaBeans в интегрированной среде разработки

Технология разработки на основе компонентов обещает позволять создавать приложения из заготовленных компонентов с написанием минимального объема программного кода. В этом разделе мы для примера попробуем создать приложение из компонентов `ImageViewerBean` и `FileNameBean`. Для начала в NetBeans 6 выберите пункт меню File⇨New Project (Файл⇨Новый проект). После этого на экране появится диалоговое окно New Project (Новый проект). Выберите в этом диалоговом окне сначала категорию Java, а затем проект Java Application (Приложение Java), как показано на рис. 8.3.

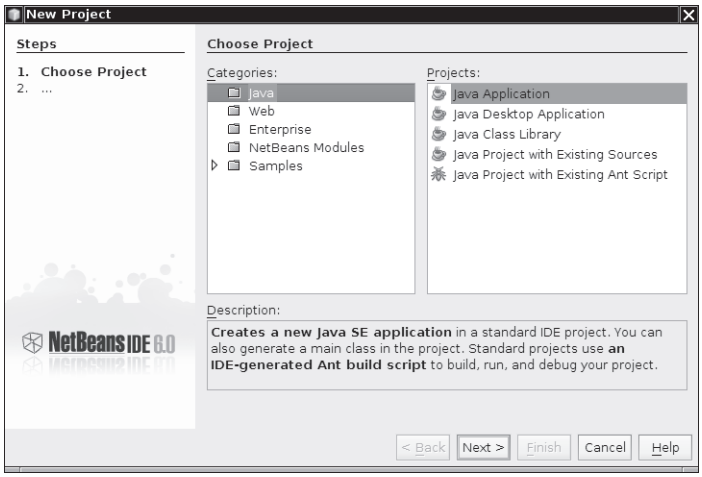


Рис. 8.3. Создание нового проекта

Сделав это, щелкните на кнопке Next (Далее). На следующем экране укажите желаемое имя для приложения (например, `ImageViewer`) и щелкните на кнопке Finish (Готово). После этого в левой части экрана появится панель просмотра проекта, а в средней — окно редактора исходного кода.

Щелкните в левой панели правой кнопкой мыши на имени проекта и в контекстном меню, которое появится после этого, выберите пункт New⇨JFrame Form (Создать⇨Форма JFrame), как показано на рис. 8.4.

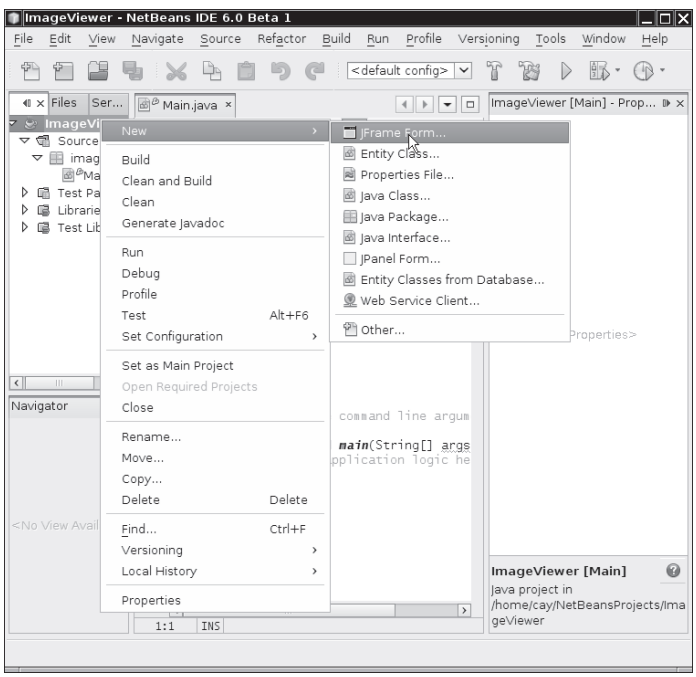


Рис. 8.4. Создание представления формы

На экране отобразится очередное диалоговое окно. Введите в нем имя реализующего фрейм класса (например, `ImageViewerFrame`) и щелкните на кнопке Finish (Готово). После этого появится редактор форм с пустым фреймом. Для того чтобы добавить компонент bean на форму, выберите его в палитре, находящейся справа от редактора форм. Затем щелкните на фрейме.

На рис. 8.5 показан результат добавления на фрейм компонента `ImageViewerBean`.

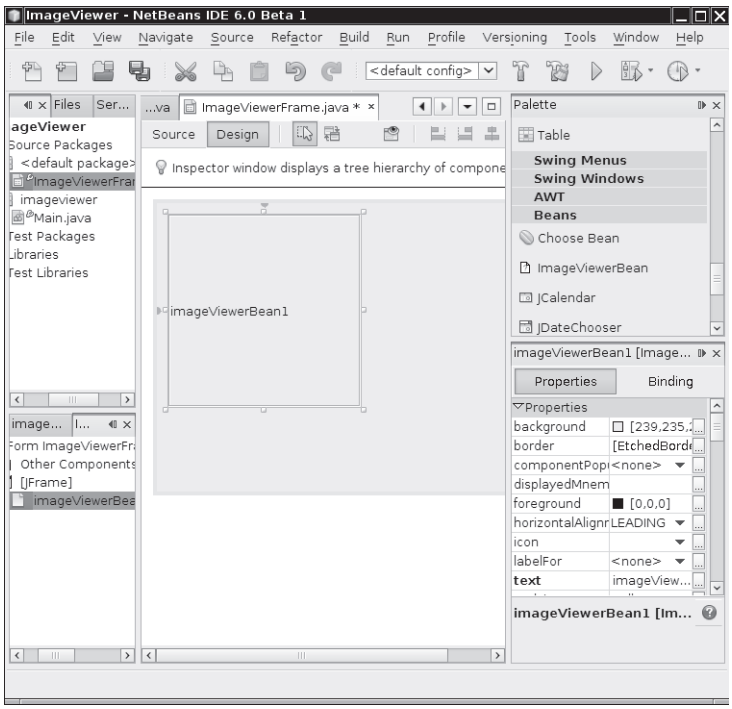


Рис. 8.5. Добавление компонента bean в состав фрейма

Если вы после этого загляните в окно редактора кода, то увидите, что теперь в исходном коде содержатся Java-инструкции для добавления bean-объектов в главную панель фрейма (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.



Интегрированная среда разработки не обязательно будет обновлять исходный код в процессе создания приложения. Она может генерировать код по окончании редактирования и, возможно, описание ваших действий будет выглядеть неожиданно для вас.

Например, экспериментальный инструмент Bean Builder, доступный по адресу: `http://bean-builder.dev.java.net`, позволяет создавать приложения с графическим пользовательским интерфейсом, вовсе не занимаясь написанием исходного кода вручную.

Компоненты JavaBeans не навязывают среде разработки принципы реализации приложения. Они лишь сообщают о себе, для того чтобы инструментальные средства могли тем или иным способом использовать полученную информацию. Из последующих разделов вы узнаете, как обеспечить при программировании компонентов доступность подобной информации для инструментов разработки.

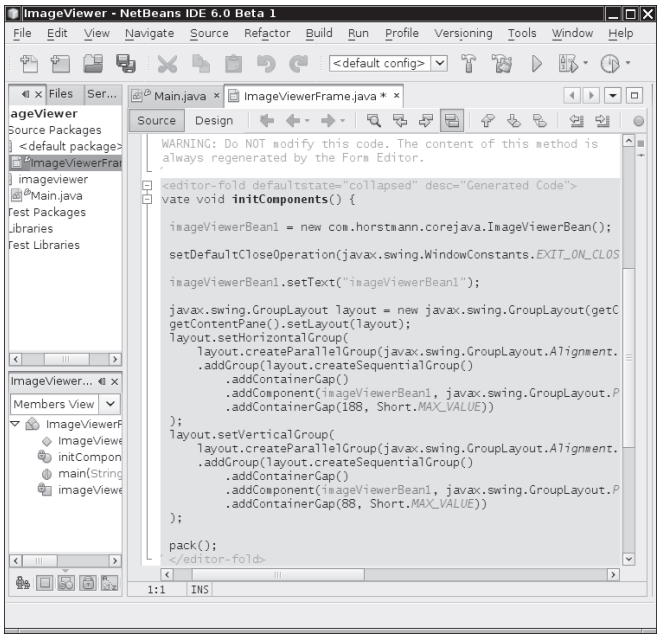


Рис. 8.6. Исходный код, необходимый для добавления компонента bean

Теперь вернитесь в окно проектирования и щелкните внутри формы на компоненте `ImageViewerBean`. В правой части окна появится панель инспектора свойств со списком имен свойств компонента JavaBeans и их текущими значениями. Инспектор свойств является чрезвычайно важной частью инструментов разработки компонентных объектов, поскольку именно путем установки свойств во время проектирования как раз и задается начальное состояние компонента.

Например, вы можете изменить свойство `text` используемой для компонента `ImageViewerBean` надписи просто путем ввода нового значения непосредственно внутри панели инспектора свойств. Изменение свойства `text` выполняется очень легко — от вас требуется лишь отредактировать строку в соответствующем текстовом поле. Попробуйте сделать это: например, измените текст надписи на `Hello`. Форма немедленно обновится и отобразит внесенное вами изменение (рис. 8.7).



При изменении значения свойства интегрированная среда разработки NetBeans сразу же обновляет исходный код соответствующим образом. Например, в случае указания для поля `text` значения `Hello`, в метод `initComponents` будет тут же добавлена инструкция `imageViewerBean.setText("Hello");`

Как уже упоминалось ранее, в других интегрированных средах разработки могут применяться другие стратегии для записи значений свойств.

Значения свойств не обязательно должны представлять собой строки, они могут быть любыми другими типами данных, допустимыми в языке Java. Для предоставления пользователям возможности выбора значения произвольного типа в интегрированных средах разработки предусмотрены специальные *редакторы свойств*. (Редакторы свойств могут быть встроены в интегрированную среду разработки или в компонент JavaBeans. Способы создания собственных редакторов свойств рассматриваются далее в главе.)

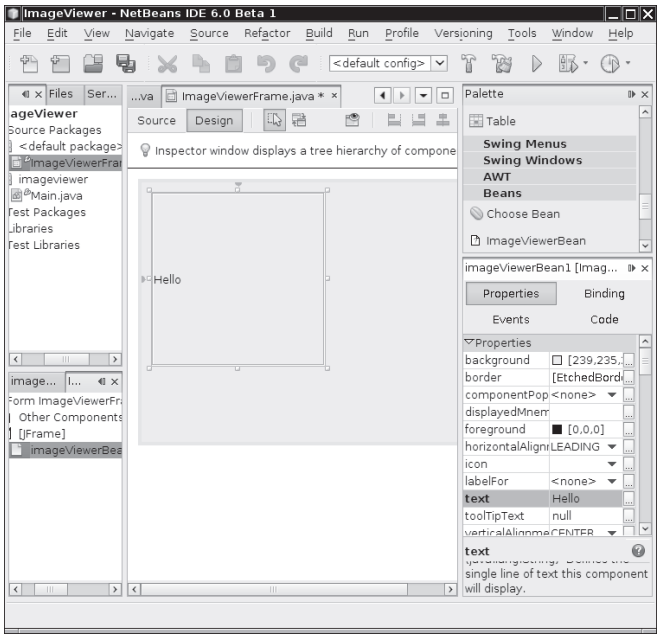


Рис. 8.7. Изменение значения свойства внутри панели инспектора свойств

В качестве примера простого редактора свойств рассмотрим свойство `foreground`, которое определяет цвет переднего плана и имеет тип `Color`. Редактор цвета предоставляет поле, содержащее строку `[0,0,0]` и кнопку с изображением троеточия (...), предназначенную для отображения диалогового окна редактирования цвета. Измените цвет переднего плана. Обратите внимание, что цвет текста изменится сразу же, как только будет модифицировано значение свойства.

Однако гораздо интереснее поэкспериментировать с выбором файла, содержащего изображение. Введите имя графического файла в соответствующем поле редактора свойств, после чего в компоненте `ImageViewerBean` сразу же появится изображение.



Если вы внимательно посмотрите на панель инспектора свойств в NetBeans, вы обнаружите там огромное количество загадочных свойств вроде `focusCycleRoot` и `paintingForPrint`. Эти свойства наследуются от суперкласса `JLabel`. О том, как их можно подавлять из панели инспектора свойств, будет рассказываться чуть позже в этой главе.

Чтобы завершить наше демонстрационное приложение, поместите внутрь фрейма объект `FileNameBean`. Нам нужно, чтобы в случае изменения свойства `fileName` этого объекта загружался новый рисунок. Обеспечить такое поведение можно с помощью события `PropertyChangeEvent`. Более подробно о нем и других подобных событиях речь пойдет позже в этой главе.

Для обеспечения реакции на это событие выберите компонент `FileNameBean` и затем перейдите в окне его инспектора свойства на вкладку Events (События). Далее щелкните на кнопке с изображением троеточия (...) рядом с пунктом `propertyChange`. После этого на экране появится диалоговое окно, показывающее, что на текущий момент с данным событием пока не ассоциируется ни одного обработчика. Щелкните в этом окне на кнопке Add (Добавить). После этого появится приглашение ввести имя метода (рис. 8.8). Введите в нем имя `loadImage`.

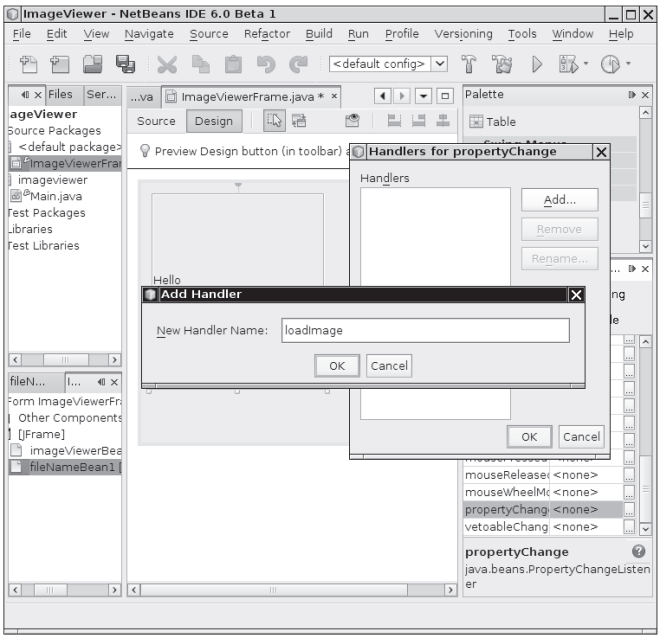


Рис. 8.8. Добавление события к компоненту JavaBeans

Теперь взгляните на окно редактора кода. Там должен появиться код обработки событий и новый метод:

```
private void loadImage(java.beans.PropertyChange evt)
{
   // TODO add your handling code here/добавьте здесь свой код обработки
}
```

Вставьте в этом методе следующую строку кода:

```
imageViewerBean1.setFileName(fileNameBean1.getFileName());
```

Затем откомпилируйте и запустите класс фрейма. В вашем распоряжении появится готовое приложение, предназначенное для просмотра изображений. Щелкните на кнопке с изображением троеточия и выберите файл. Выбранное изображение появится в окне просмотра (рис. 8.9).



Рис. 8.9. Приложение для просмотра изображений

Этот процесс демонстрирует способ создания Java-приложений на основе компонентов JavaBeans путем указания их свойств и создания небольших фрагментов кода, предназначенного для обработки событий.

Соглашения об именовании свойств и событий компонента JavaBeans

В данном разделе рассматриваются основные правила создания новых компонентов. Прежде всего, необходимо подчеркнуть, что для JavaBeans *не* существует никакого особого класса, который мог бы выступить в качестве суперкласса компонента и определять его поведение. Все визуальные компоненты JavaBeans прямо или косвенно унаследованы от класса `Component`, а невизуальные компоненты обычно формируются как подклассы класса `Object`. Напомним, что компонент JavaBeans представляет собой *произвольный* класс, которым можно управлять из интегрированной среды разработки. Для определения природы компонента интегрированная среда разработки анализирует не суперкласс, а имена методов. Поэтому имена методов компонента bean должны удовлетворять определенным соглашениям.



Существует класс `java.beans.Beans`, но все его методы являются статическими. Следовательно, бессмысленно создавать его подклассы. К тому же нельзя организовать класс, который наследовал бы и `Beans` и `Components` одновременно, поэтому визуальные компоненты JavaBeans не могут воспользоваться этим подходом. Единственное реальное применение класса `Beans` — проверка взаимодействия со средой интегрированной разработки.

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 способами. Если автор компонента следует стандартным соглашениям об именовании, то интегрированная среда разработки может использовать для определения нужных свойств и событий JavaBeans механизм рефлексии. Кроме того, автор компонента bean может использовать специальный *информационный класс компонента* (bean information class), который содержит описание свойств и событий, необходимых интегрированной среде разработки. Начнем с более простого способа – использования соглашения об именах, а затем рассмотрим применение информационных классов.



Хотя в документации стандартные соглашения об именах называются шаблонами проектирования (design patterns), на самом деле они не имеют ничего общего с шаблонами проектирования, которые применяются в объектно-ориентированном программировании.

Соглашения об именах для свойств имеют очень простую структуру. Например, метод `public` *`Тип`* `get`*`Имя_Свойства`*`()` возвращает (читает) значение свойства указанного типа, а метод `public void set`*`Имя_Свойства`*`(`*`Тип значение`*`)` устанавливает (записывает) значение для этого свойства.

Например, компонент `ImageViewerBean` содержит только одно свойство, допускающее чтение и запись. Это свойство содержит имя файла и для него используются следующие методы:

```
public String getFileName()
public void setFileName(String newValue)
```

Свойство может быть определено только для чтения, если для него предусмотрен get-метод, но отсутствует set-метод.



Созданные get- и set-методы, как и любые другие методы Java, могут выполнять произвольные действия, а не только возвращать и задавать значения поля. Например, метод setFileName() класса ImageVieverBean не только задает значение поля fileName, но также открывает файл и загружает изображение.



В Visual Basic для работы со свойствами также используются get- и set-методы, а в Delphi — методы read и write. Но в обоих языках явно указывается определение свойства, так что компилятор может просто определить намерения программиста. Средства работы со свойствами, применяемые в этих языках, имеют еще одно преимущество: указание имени свойства в левой части выражения автоматически связано с вызовом set-метода, а указание имени свойства в самом выражении автоматически связано с вызовом get-метода. Например, в Visual Basic можно использовать код imageBean.fileName = "corejava.gif"; вместо кода imageBean.setFileName("corejava.gif");.

При создании языка Java рассматривался вопрос о целесообразности подобного подхода, однако разработчики посчитали, что не следует скрывать имена методов за языковыми конструкциями, напоминающими обычное присваивание значений переменным.

В соглашении об именах языка Java для свойств логического типа вместо префиксов get/set нужно использовать префиксы is/set так, как показано ниже:

```
public boolean isИмя_Свойства();
public void setИмя_Свойства(boolean b);
```

Например, компонент JavaBeans с анимационными возможностями может иметь свойство running с двумя методами:

```
public boolean isRunning();
public void setRunning(boolean b);
```

Таким образом, метод setRunning() может запускать и останавливать анимацию, а метод isRunning() — сообщать о текущем состоянии анимации.



Для доступа к логическим свойствам допустимо использовать методы с префиксом get, однако префикс is предпочтительнее.

Создавая методы, обращайте внимание на использование прописных букв в их именах. Так, рассматриваемое здесь в качестве примера свойство должно иметь имя fileName (буква f — строчная), а в методах get и set указывается прописная буква F (getFileName(), setFileName()). Для определения имени свойства анализатор компонентов использует *процедуру приведения к нижнему регистру* (decapitalization). (В нижний регистр преобразуется первый символ, следующий за префиксом get или set.) Благодаря такому подходу формируются имена свойств и методов, привычные для большинства программистов.

Если же прописными являются две первые буквы свойства (такому свойству, например, соответствует метод getURL()), регистр первой буквы не преобразуется вовсе. И действительно, свойство с именем uRL выглядело бы, по меньшей мере, странно.



Может возникнуть вопрос: что делать, если в классе есть методы `get` и `set`, не имеющие отношения к свойствам, доступным пользователям посредством инспектора? Если вы создаете класс с нуля, то лучше всего придумать для методов другие имена. Однако может случиться так, что методы с подобными именами наследуются от суперкласса. Например, если компонент создается как подкласс `JPanel` или `JLabel`, инспектор отобразит очень большое количество свойств. Далее в главе вы познакомитесь с *информацией о компоненте*, которая дает возможность управлять процессом распознавания свойств. В частности, вы сможете явно указывать, какие свойства должны быть представлены пользователю интегрированной системы.

Соглашение об именах для событий еще проще. Интегрированная среда разработки определяет события, генерируемые компонентом JavaBeans, при наличии методов включения и удаления обработчиков. Имена классов событий должны оканчиваться суффиксом `Event`, а сами классы должны расширять класс `EventObject`.

Предположим, что компонент JavaBeans генерирует событие типа *Имя_События*`Event`. В таком случае интерфейс обработчика (слушателя) должен называться *Имя_События*`Listener`, а методы для управления слушателями — так, как показано ниже:

```
public void add Имя_События Listener (EventNameListener e);
public void remove Имя_События Listener (EventNameListener e);
public Имя_события Listener get Имя_события Listeners()
```

Если вы посмотрите на код `ImageViewerBean`, то увидите, что в нем события не предусмотрены. Однако многие компоненты Swing генерируют события, поэтому содержат имена, соответствующие рассматриваемым соглашениями. Например, класс `AbstractButton` генерирует объекты `ActionEvent` и содержит следующие методы для управления объектами `ActionListener`:

```
public add ActionListener(ActionListener e)
public void remove ActionListener(ActionListener e)
ActionListener[] getActionListeners()
```



Даже если ваш класс события не является подклассом `EventObject`, есть шансы, что он будет скомпилирован, поскольку ни один из методов `EventObject` реально не требуется. Однако не исключено возникновение необъяснимой ошибки, так как механизм интроспекции не распознает события.

Типы свойств компонентов JavaBeans

Сложный компонент JavaBeans будет иметь множество различных свойств и событий. Эти свойства могут как совсем простыми, как, например, свойство `fileName`, которое было у компонента `ImageViewerBean` и компонента `FileNameBean`, так и очень сложными, как, например, палитра цветов или массив информационных точек — оба этих случая будут более подробно рассматриваться позже в этой главе. Спецификация JavaBeans позволяет использовать свойства четырех типов, каждый из которых описывается в следующих разделах.

Простые свойства

Это свойства, которые содержат одно значение, например, строку или число. Примером простого свойства является свойство `fileName` компонента `ImageViewerBean`.

Простое свойство довольно легко программируется с помощью описанного выше соглашения об именовании. Например, как показано в листинге 8.1, для создания простого свойства типа `String` достаточно создать представленные ниже методы `set` и `get`:

```
public void setFileName ( String f )
{
   fileName = f;
   image = ...
   repaint();
}

public String getFileName ( String f )
{
   if (file == null) return "";
   else return file.getPath();
}
```

Индексированные свойства

Индексированное свойство позволяет задавать массив. В случае использования индексированного свойства требуется предоставлять две пары методов `get` и `set`: одну для массива и одну для отдельных элементов. Выглядеть они должны следующим образом:

```
Тип[] getИмя_Свойства();
void setИмя_Свойства(Тип[] новое_значение);
Тип getИмя_Свойства(int i);
void setИмя_Свойства(int i, Тип новое_значение);
```

Например, в компоненте `FileNameBean` индексированное свойство используется для расширений файлов и состоит из четырех следующих методов:

```
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;
}
. . .
private String[] extensions;
```

Использовать метод `setИмя_Свойства(int, Type)` для *расширения* массива нельзя. Для этого нужно вручную создавать новый массив и затем передавать его методу `setИмя_Свойства(Тип[])`.

Связанные свойства

Связанные свойства уведомляют обработчиков об изменении их значения. Например, свойство `fileName` компонента `FileNameBean` является примером такого связанного свойства. При изменении имени файла компонент `ImageViewerBean` автоматически уведомляется об этом и загружает новый файл.

Для создания связанного свойства необходимо обеспечить следующее.

1. При изменении значения свойства компонент JavaBeans должен посылать всем зарегистрированным обработчикам событие `PropertyChangeEvent`. Такое изменение может происходить при вызове `set`-метода или изменении значения каким-нибудь методом (например, слушателем действий кнопки с изображением троеточия).
2. Для предоставления обработчикам (слушателям) возможности автоматически регистрировать самих себя компонент JavaBeans должен реализовать два следующих метода:

   ```
   void addPropertyChangeListener(PropertyChangeListener listener)
   void removePropertyChangeListener(PropertyChangeListener listener)
   ```

   Также рекомендуется (но не требуется) предоставлять еще и такой метод:

   ```
   PropertyChangeListener[] getPropertyChangeListeners()
   ```

В пакете `java.beans` содержится полезный класс `PropertyChangeSupport`, умеющий самостоятельно (без участия разработчика) управлять работой обработчиков событий. Чтобы использовать его, нужно сначала добавить поле экземпляра такого класса:

```
private PropertyChangeSupport changeSupport = new PropertyChangeSupport(this);
```

Далее необходимо поручить выполнение операций по добавлению и удалению обработчиков изменений свойств этому объекту следующим образом:

```
public void addPropertyChangeListener(
   PropertyChangeListener listener)
{
   changeSupport.addPropertyChangeListener(listener);
}

}

public PropertyChangeListener[] getPropertyChangeListeners()

{

  return changeSupport.getPropertyChangeListeners();

}
```

После этого с помощью метода `firePropertyChange()` объекта `PropertyChangeSupport` нужно сделать так, чтобы при каждом изменении значения свойства, соответствующее событие передавалось всем зарегистрированным обработчикам. Этот метод принимает три параметра: имя свойства, прежнее значение и новое значение. Ниже приведен стандартный код типичного `set`-метода связанного свойства:

```
public void setValue(Тип newValue)
{
  Type oldValue = getValue();
  value = newValue;
  changeSupport.firePropertyChange("propertyName", oldValue, newValue);
}
```

Для уведомления об изменении индексированного свойства используется следующий вызов:

```
changeSupport.fireIndexedPropertyChange("propertyName", index, oldValue, newValue);
```



Если компонент JavaBeans расширяет какой-нибудь класс, который, в конечном счете, расширяет класс `JComponent`, тогда реализовывать методы `addPropertyChangeListener()`, `removePropertyChangeListener()` и `getPropertyChangeListeners()` не нужно, поскольку они уже реализованы в суперклассе `JComponent`. Для уведомления обработчиков событий изменения свойства необходимо просто вызывать метод `firePropertyChange()` суперкласса `JComponent`. Генерация уведомлений об изменении индексированных свойств, к сожалению, не поддерживается.

Другие компоненты JavaBeans, которые требуется уведомлять об изменении значений свойств, должны иметь интерфейс `PropertyChangeListener`. Этот интерфейс включает только один метод:

```
void propertyChange(PropertyChangeEvent event);
```

Объект `PropertyChangeEvent` содержит имя свойства, а также его старое и новое значения, получаемые с помощью методов `getPropertyName`, `getOldValue` и `getNewValue`.

Если тип свойства не является типом класса, тогда объекты значений этого свойства представляют собой экземпляры обычных классов-оболочек.

Свойства с ограничениями

Термин *с ограничениями* означает, что *любой* обработчик может запрещать предлагаемые изменения и восстанавливать предыдущие установки. В библиотеке Java можно найти лишь несколько примеров свойств с ограничениями. Одним из них является свойство `closed` класса `JInternalFrame`. При попытке вызова метода `setClosed(true)` для внутреннего фрейма оповещаются все обработчики `VetoableChangeListener`. Если хотя бы один из них выдает исключение `PropertyVetoException`, свойство `closed` *не* изменяется, а метод `setClosed()` генерирует точно такое же исключение. В частности, `VetoableChangeListener` может запрещать закрытие фрейма в случае, если его содержимое еще не было сохранено.

Для создания свойства с ограничениями необходимо, чтобы у компонента JavaBeans было два следующих метода для управления объектами `VetoableChangeListener`:

```
public void addVetoableChangeListener(VetoableChangeListener listener);
public void removeVetoableChangeListener(VetoableChangeListener listener);
```

Помимо этого, у него еще также должен быть и следующий метод для получения всех обработчиков (слушателей):

```
VetoableChangeListener[] getVetoableChangeListeners()
```

Подобно вспомогательному классу, предназначенному для управления обработчиками событий изменения свойств, существует вспомогательный класс `VetoableChangeSupport`, управляющий обработчиками изменений, которые могут быть запрещены. Ваш компонент должен содержать экземпляр этого класса:

```
private VetoableChangeSupport vetoSupport =
   new VetoableChangeSupport(this);
```

Добавление и удаление обработчиков делегируются данному объекту. Пример его использования приведен ниже.

```
public void addVetoableChangeListener(VetoableChangeListener listener)
{
   vetoSupport.addVetoableChangeListener(listener);
}
```

```
public void removeVetoableChangeListener(VetoableChangeListener listener)
{
   vetoSupport.removeVetoableChangeListener(listener);
}
```

Таким образом, обновление значения свойства с ограничениями включает в себя три этапа.

1. Оповещение обработчиков изменений свойств с ограничениями о *намерении* изменить значение. (Для этого используется метод `fireVetoableChange()` класса `VetoableChangeSupport`.)
2. Если ни один из обработчиков не сгенерирует исключение `PropertyVetoException`, значение свойства обновляется.
3. Передача всем обработчикам подтверждения о том, что изменения выполнены.

Соответствующий пример кода представлен ниже.

```
public void setValue(Type newValue) throws PropertyVetoException
{
   Type oldValue = getValue();
   vetoSupport.fireVetoableChange("value", oldValue, newValue);
   // Если следующая строка кода получит управление,
   // значит, запрета на изменение значения не было.
   value = newValue;
   changeSupport.firePropertyChange("value", oldValue, newValue);
}
```

Обратите внимание, что до тех пор, пока все зарегистрированные обработчики не согласятся с предложенными изменениями, значение свойства остается прежним. С другой стороны, ни один обработчик не может быть уверенным, что изменения, против которых он не возражает, действительно будут выполнены. Единственный способ убедиться в том, что значение свойства обновлено, — использовать обработчик события, соответствующего изменению значения свойства.



Если компонент JavaBeans расширяет класс `JComponent`, отдельный объект `VetoableChangeSupport` не нужен. Вместо него следует просто вызывать метод `fireVetoableChange` суперкласса `JComponent`. Обратите внимание на то, что устанавливать в `JComponent` обработчика запрещаемых изменений для конкретного свойства нельзя. Обрабатываться должны обязательно все подобные изменения.

Завершается наше рассмотрение свойств JavaBeans демонстрацией полного кода для компонента `FileNameBean` (листинг 8.2). Этот компонент обладает индексированным свойством `extension` и предполагающим наличие ограничений свойством `filename`. Поскольку он расширяет класс `JPanel`, объект `PropertyChangeSupport` явно не используется. Вместо этого применяются предусмотренные в классе `JPanel` возможности для управления обработчиками событий.

Листинг 8.2. Содержимое файла `FileNameBean.java`

```
1.  package com.horstmann.corejava;
2.  import java.awt.*;
3.  import java.awt.event.*;
4.  import java.io.*;
5.  import java.util.*;
6.  import javax.swing.*;
7.  import javax.swing.filechooser.*;
```

```
8.  /**
9.    *Компонент, позволяющий задавать имена файлов.
10.   * @version 1.30 2007-10-03
11.   * @author Cay Horstmann
12.   */
13. public class FileNameBean extends JPanel
14. {
15.    public FileNameBean()
16.    {
17.       dialogButton = new JButton("...");
18.       nameField = new JTextField(30);
19.
20.       chooser = new JFileChooser();
21.       setPreferredSize(new Dimension(XPREFSIZE, YPREFSIZE));
22.
23.       setLayout(new GridBagLayout());
24.       GridBagConstraints gbc = new GridBagConstraints();
25.       gbc.weightx = 100;
26.       gbc.weighty = 100;
27.       gbc.anchor = GridBagConstraints.WEST;
28.       gbc.fill = GridBagConstraints.BOTH;
29.       gbc.gridwidth = 1;
30.       gbc.gridheight = 1;
31.       add(nameField, gbc);
32.       dialogButton.addActionListener(new ActionListener()
33.        {
34.          public void actionPerformed(ActionEvent event)
35.         {
36.           chooser.setFileFilter(new FileNameExtensionFilter(
37.                                    Arrays.toString(extensions),extensions));
38.           int r = chooser.showOpenDialog(null);
39.           if (r == JFileChooser.APPROVE_OPTION)
40.           {
41.             File f = chooser.getSelectedFile();
42.             String name = f.getAbsolutePath();
43.              setFileName(name);
44.           }
45.         }
46.       });
47.       nameField.setEditable(false);
48.
49.       gbc.weightx = 0;
50.       gbc.anchor = GridBagConstraints.EAST;
51.       gbc.fill = GridBagConstraints.NONE;
52.       gbc.gridx = 1;
53.       add(dialogButton, gbc);
54. }
55.    /**
56.     * Установка свойства fileName.
57.     * @param newValue Новое имя файла
58.     */
59.    public void setFileName(String newValue)
60.    {
61.       String oldValue = nameField.getText();
62.       nameField.setText(newValue);
63.       firePropertyChange("fileName", oldValue, newValue);
64.    }
```

```
65.    /**
66.     * Получение свойства fileName.
67.     * @return Имя выбранного файла
68.     */
69.    public String getFileName()
70.    {
71.       return nameField.getText();
72.    }
73.
74.    /**
75.     * Получение свойства Extensions.
76.     * @return Расширения по умолчанию в окне выбора файлов
77.     */
78.    public String getExtensions()
79.    {
80.       return Extensions;
81.    }
82.
83.   /**
84.    * Установка свойства Extensions.
85.    * @param newValue Новые расширения по умолчанию
86.    */
87.   public void setExtensions(String[] newValue)
88.   {
89.     extensions = newValue;
90.   }
91.
92.   /**
93.    * Получение одного из значений свойства Extensions.
94.    * @param i Индекс требующегося значения свойства
95.    * @return Значение, находящееся по указанному индексу
96.    */
97.   public String getExtensions(int i)
98.   {
99.     if (0 <= i && i < extensions.length) return extensions[i];
100.    else return "";
101.  }
102.
103.  /**
104.   * Установка одного из значений свойства Extensions.
105.   * @param i Индекс требуюшегося значения свойства
106.   * @param newValue Новое значение, находящееся по указанному индексу
107.   */
108.  public void setExtensions(int i, String newValue)
109.  {
110.    if (0 <= i && i < extensions.length) extensions[i] = newValue;
111.  }
112.
113.  private static final int XPREFSIZE = 200;
114.  private static final int YPREFSIZE = 20;
115.  private JButton dialogButton;
116.  private JTextField nameField;
117.  private JFileChooser chooser;
118.  private String[] extensions = { "gif", "png" };
119. }
```

API java.beans.PropertyChangeListener 1.1

- `void propertyChange(PropertyChangeEvent event)`

  Вызывается при генерации события изменения свойства.

API java.beans.PropertyChangeSupport 1.1

- `PropertyChangeSupport(Object sourceBean)`

  Создает объект `PropertyChangeSupport`, управляющий обработчиками событий, которые соответствуют изменению значений связанных свойств данного компонента.

- `void addPropertyChangeListener(PropertyChangeListener listener)`
- `void addPropertyChangeListener(String propertyName, PropertyChangeListener listener)` 1.2

  Регистрирует обработчик событий изменения значений всех связанных свойств или конкретного связанного свойства.

- `void removePropertyChangeListener(PropertyChangeListener listener)`
- `void removePropertyChangeListener(String propertyName, PropertyChangeListener listener)` 1.2

  Удаляет ранее зарегистрированный обработчик событий для 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

  Отправляет событие `PropertyChangeEvent` зарегистрированным обработчикам.

- `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 `IndexedPropertyChangeEvent` зарегистрированным обработчикам.

- `PropertyChangeListener[] getPropertyChangeListeners()` 1.4
- `PropertyChangeListener[] getPropertyChangeListeners(String propertyName)` 1.4

  Возвращает обработчики для всех связанных событий или только для связанного события с определенным именем.

API java.beans.PropertyChangeEvent 1.1

- `PropertyChangeEvent(Object sourceBean, String propertyName, Object oldValue, Object newValue)`

  Создает объект `PropertyChangeEvent`, описывающий изменение значения свойства с `oldValue` на `newValue`.

- `Stirng getPropertyName()`

  Возвращает имя свойства.

- `Object getNewValue()`

  Возвращает новое значение свойства.

- `Object getOldValue()`

  Возвращает прежнее значение свойства.

API java.beans.IndexedPropertyChangeEvent 5.0

- `IndexedPropertyChangeEvent(Object sourceBean, String propertyName, int index, Object oldValue, Object newValue)`

  Создает новый объект `IndexedPropertyChangeEvent`, описывающий изменение значения данного свойства с `oldValue` на `newValue`.

- `int getIndex()`

  Возвращает индекс, соответствующий изменению значения.

API java.beans.VetoableChangeListener 1.1

- `void vetoableChange(PropertyChangeEvent event)`

  Вызывается при намерении изменить свойство. Если изменения не допустимы, генерируется исключение `PropertyVetoException`.

API java.beans.VetoableChangeSupport 1.1

- `VetoableChangeSupport(Object sourceBean)`

  Создает объект `PropertyChangeSupport`, который управляет обработчиками изменения свойств с ограничениями для данного компонента.

- `void addVetoableChangeListener(VetoableChangeListener listener)`
- `void addVetoableChangeListener(String propertyName, VetoableChangeListener listener)` 1.2

  Регистрирует обработчик изменений для всех свойств с ограничениями или для свойства с конкретным именем.

- `void removeVetoableChangeListener(VetoableChangeListener listener)`
- `void removeVetoableChangeListener(String propertyName, VetoableChangeListener listener)` 1.2

  Удаляет ранее зарегистрированный обработчик изменений для свойств с ограничениями.

- `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

  Передает объект `VetoableChangeEvent` зарегистрированным обработчикам.

- `VetoableChangeListener[] getVetoableChangeListeners()` 1.4
- `VetoableChangeListener[] getVetoableChangeListeners(String propertyName)` 1.4

Возвращает обработчики изменений для всех свойств с ограничениями или для свойства с конкретным именем.

API java.awt.Component 1.0

- `void addPropertyChangeListener(PropertyChangeListener listener)` 1.2
- `void addPropertyChangeListener(String propertyName, PropertyChangeListener listener)` 1.2

Регистрирует обработчик событий изменения значения для всех связанных свойств или только для связанного свойства с указанным именем.

- `void removePropertyChangeListener(PropertyChangeListener listener)` 1.2
- `void removePropertyChangeListener(String propertyName, PropertyChangeListener listener)` 1.2

Удаляет зарегистрированный ранее обработчик событий изменения значений.

- `void firePropertyChange(String propertyName, Object oldValue, Object newValue)` 1.2

Посылает зарегистрированным обработчикам событие `PropertyChangeEvent`.

API javax.swing.JComponent 1.2

- `void addVetoableChangeListener(VetoableChangeListener listener)`

Регистрирует обработчик событий для всех свойств с ограничениями или для свойства с указанным именем.

- `void removeVetoableChangeListener(VetoableChangeListener listener)`

Удаляет зарегистрированный ранее обработчик событий для свойства с ограничениями.

- `void fireVetoableChange(String propertyName, Object oldValue, Object newValue)`

Посылает зарегистрированным обработчикам событие `VetoableChangeEvent`.

API java.beans.PropertyVetoException 1.1

- `PropertyVetoException(String message, PropertyChangeEvent event)`

Создает новое исключение `PropertyVetoException`.

- `PropertyChangeEvent getPropertyChangeEvent()`

Возвращает событие `PropertyChangeEvent`, на которое был наложен запрет.

Классы BeanInfo

При использовании стандартных соглашений по именованию методов интегрированная среда разработки имеет возможность применять рефлексию для распознавания свойств и событий. Для тех, кто только приступает к разработке компонентов, такой подход кажется простым, однако в дальнейшем становятся ясны его недостатки. По мере усложнения компонента увеличивается число его свойств; не все они

должны быть видны пользователю интегрированной системы. Более того, в составе компонента могут присутствовать пары методов get/set, не имеющих отношения к свойствам bean.

При необходимости иметь более гибкий механизм для предоставления информации о компоненте JavaBeans, рекомендуется определять объект, реализующий интерфейс BeanInfo. В случае предоставления такого объекта инструменты, входящие в состав среды разработки, за всеми сведениями о поддерживаемых компонентом возможностях будут обращаться к нему.

Имя класса с информацией о компоненте bean должно получаться путем добавления к имени компонента суффикса BeanInfo. Например, класс информации о bean-компоненте, ассоциируемый с классом ImageViewerBean, должен называться ImageViewerBeanBeanInfo. Кроме того, этот класс еще также должен и входить в состав того же самого пакета, что и сам компонент.

Как правило, разработчикам не приходится реализовывать все методы, объявленные в интерфейсе BeanInfo. Вместо этого можно создать подкласс уже существующего вспомогательного класса SimpleBeanInfo, в котором реализованы все методы BeanInfo.

Чаще всего класс BeanInfo создают для контроля свойств компонента. Для каждого свойства формируется объект PropertyDescriptor, конструктору которого передают имя свойства и класс, содержащий его:

```
PropertyDescriptor descriptor = new PropertyDescriptor("fileName",
  ImageViewerBean.class);
```

Кроме того, в классе BeanInfo необходимо реализовать метод getPropertyDescriptors(), возвращающий массив всех дескрипторов свойств.

Предположим, например, что разработчик класса ImageViewerBean хочет спрятать от пользователя интегрированной среды все свойства, унаследованные от суперкласса Jlabel, и предоставить ему только свойство fileName. Сделать это позволит приведенный ниже класс BeanInfo.

```
// Класс BeanInfo для компонента ImageViewerBean
class ImageViewerBeanBeanInfo extends SimpleBeanInfo
{
  public PropertyDescriptor[] getPropertyDescriptors()
  {
    return propertyDescriptors;

   }

   private PropertyDescriptor[] propertyDescriptors = new PropertyDescriptor[]
     {
       new PropertyDescriptor("fileName",
         ImageViewerBean.class);
     };
  }
}
```

Существуют также методы, возвращающие массивы EventSetDescriptor и MethodDescriptor, но они применяются значительно реже. Если один из этих методов возвращает значение null (именно так реализован класс SimpleBeanInfo), то используются стандартные соглашения об именовании. Однако если вы переопределите метод так, чтобы он возвращал значение, отличное от null, вам придется включить в массив все свойства, события или методы.



В некоторых случаях может понадобиться универсальный код для распознавания свойств или событий произвольного компонента. Вызовите статический метод `getBeanInfo()` класса `Introspector`. Класс `Introspector` формирует класс `BeanInfo`, который полностью описывает компонент, принимая во внимание информацию в сопутствующем классе.

Заслуживает внимания также метод `getIcon()`, объявленный в интерфейсе `BeanInfo`. Он позволяет связать с компонентом пиктограмму, предоставленную пользователем. Инструменты среды разработки отображают эту пиктограмму на панели инструментов. Можно задавать четыре отдельные битовые карты. В интерфейсе `BeanInfo` объявлены четыре константы, определяющие стандартные размеры пиктограмм:

```
ICON_COLOR_16x16
ICON_COLOR_32x32
ICON_MONO_16x16
ICON_MONO_32x32
```

Ниже приведен пример использования вспомогательного метода `loadImage()` в классе `SimpleBeanInfo` для загрузки изображений пиктограмм:

```
public class ImageViewerBeanBeanInfo extends SimpleBeanInfo
{
  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;
  }
  private Image iconColor16;
  private Image iconColor32;
  private Image iconMono16;
  private Image iconMono32;
}
```

API java.beans.Introspector 1.1

- `static BeanInfo getBeanInfo(Class<?> beanClass)`

  Возвращает информацию о компоненте для указанного класса.

API java.beans.BeanInfo 1.1

- `PropertyDescriptor[] getPropertyDescriptors()`

  Возвращает дескрипторы свойств bean-компонента. Возврат значения `null` указывает на то, что для поиска свойств должны использоваться соглашения об именовании.

- `Image getIcon(int iconType)`

  Возвращает объект `Image`, который может представлять bean-компонент в панелях инструментов и других подобных элементах пользовательского интерфейса. Как уже упоминалось выше, для стандартных типов пиктограмм предусмотрены четыре константы.

API java.beans.SimpleBeanInfo 1.1

- `Image loadImage(String resourceName)`

  Возвращает файл объекта изображения, ассоциируемый с данным ресурсом. Имя ресурса представляет собой путь, взятый относительно каталога, содержащего класс информации о bean-компоненте.

API java.beans.FeatureDescriptor 1.1

- `String getName()`
- `void setName(String name)`

  Возвращает или устанавливает имя элемента, используемое в программе.

- `String getDisplayName()`
- `void setDisplayName(String displayName)`

  Возвращает или устанавливает отображаемое имя элемента. По умолчанию принимается имя, возвращаемое методом `getName()`. В настоящее время поддержка имен элементов на различных языках отсутствует.

- `String getShortDescription()`
- `void setShortDescription(String text)`

  Возвращает или устанавливает строку, которую инструменты интегрированной среды могут использовать в качестве краткого описания элемента. По умолчанию принимается значение, возвращаемое методом `getDisplayName()`.

- `boolean isExpert()`
- `void setExpert(boolean b)`

  Возвращает или устанавливает флаг, который интегрированная среда разработки может использовать, принимая решение о предоставлении начинающему пользователю доступа к элементу.

- `boolean isHidden()`
- `void setHidden(boolean b)`

  Возвращает или устанавливает флаг, сообщающий о том, что интегрированная среда разработки не должна отображать этот элемент.

API java.beans.PropertyDescriptor 1.1

- `PropertyDescriptor(String propertyName, Class<?> beanClass)`
- `PropertyDescriptor(String propertyName, Class<?> beanClass, String getMethod, String setMethod)`

  Создает объект `PropertyDescriptor`. Если в процессе интроспекции возникает ошибка, эти методы генерируют исключение `IntrospectionException`. Первый из указанных конструкторов предполагает, что для `get`- и `set`-методов использовались стандартные соглашения об именовании.

- `Class<?> getPropertyType()`

  Возвращает объект `Class` для типа свойства.
- `Method getReadMethod()`
- `Method getWriteMethod()`

  Возвращает метод для получения или установки свойства.

API **java.beans.IndexedPropertyDescriptor 1.1**

- `IndexedPropertyDescriptor(String propertyName, Class<?> beanClass)`
- `IndexedPropertyDescriptor(String propertyName, Class<?> beanClass, String getMethod, String setMethod, tring indexedGetMethod, String indexedSetMethod)`

  Создает объект `IndexedPropertyDescriptor` для индексированного свойства. Первый из указанных конструкторов предполагает следование стандартным соглашениям при именовании `get`- и `set`-методов.
- `Method getIndexedReadMethod()`
- `Method getIndexedWriteMethod()`

  Возвращает метод для получения или установки индексированного значения в свойстве.

Редакторы свойств

При добавлении в компонент свойства целочисленного типа или строки оно автоматически отобразится в инспекторе свойств. Что же произойдет при добавлении свойства более сложного типа, значения которого не удается изменить в поле редактирования, например, даты или цвета? Для этого потребуется создать специальный компонент, с помощью которого пользователь смог бы модифицировать значения этих свойств. Такие компоненты называются *редакторами свойств* (property editors). Например, редактор свойства-даты может быть представлен в виде календаря, в котором пользователь выбирает дату. Редактор свойств для изменения цвета должен предоставить пользователю возможность задавать красную, зеленую и синюю составляющие.

На самом деле в интегрированной среде разработки NetBeans есть редактор свойств для редактирования цветов. В нем также имеются редакторы свойств для объектов базовых типов, таких как `String` (текстовое поле) и `boolean` (флажок опции).

Для того чтобы предоставить пользователю новый редактор свойств, необходимо выполнить ряд действий. Сначала следует создать информационный класс для компонента. Для этого нужно переопределить метод `getPropertyDescriptors()`, который возвращает массив объектов `PropertyDescriptor`. Необходимо создать по одному такому объекту для каждого свойства, которое должно отображаться в окне редактора, *даже для тех свойств, которые используют предлагаемые по умолчанию редакторы*:

```
PropertyDescriptor descriptor
  = new PropertyDescriptor("titlePosition", ChartBean.class);
```

Затем следует использовать метод `setPropertyEditorClass` класса `PropertyDescriptor`:

```
descriptor.setPropertyEditorClass(TitlePositionEditor.class);
```

Далее нужно создать массив описаний для свойств вашего компонента. Например, компонент `ChartBean`, который рассматривается в этом разделе, имеет 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 код класса `ChartBeanBeanInfo`, в котором определены редакторы для этих свойств.

1. Метод `getPropertyDescriptors()` возвращает дескриптор каждого свойства. Для свойств `title` и `graphColor` используются редакторы по умолчанию. Это средства для редактирования строки и выбора цвета, поставляемые в составе инструментальной среды разработки.
2. Для свойств `titlePosition`, `values` и `inverse` применяются специальные редакторы. Их типы соответственно `TitlePositionEditor`, `DoubleArrayEditor` и `InverseEditor`.

На рис. 8.10 показан внешний вид компонента `ChartBean`. Вверху отображается заголовок (за который отвечает свойство `title`). Он может размещаться слева, справа или по центру. За значения в графах диаграммы отвечает свойство `values`. Если свойство `inverse` имеет значение `true`, то фон становится цветным, а столбцы диаграммы — белыми. Код этого компонента можно найти в загружаемом коде для этой книги: по сути, он представляет собой просто видоизмененный аплет диаграммы из главы 10 первого тома.

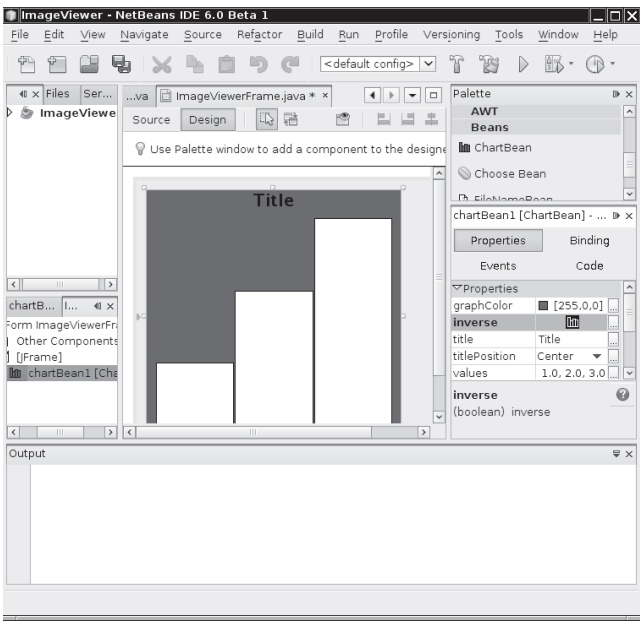


Рис. 8.10. Компонент `ChartBean`

Листинг 8.3. Содержимое файла `ChartBeanBeanInfo.java`

```
1.  package com.horstmann.corejava;
2.
3.  import java.awt.*;
4.  import java.beans.*;
5.
6.  /**
7.   * Информация о компоненте ChartBean;
8.   * определение редакторов свойств.
9.   * @version 1.20 2007-10-05
10.  * @author Cay Horstmann
11.  */
12. public class ChartBeanBeanInfo extends SimpleBeanInfo
13. {
14.   public ChartBeanBeanInfo()
15.   {
16.     iconColor16 = loadImage("ChartBean_COLOR_16x16.gif");
17.     iconColor32 = loadImage("ChartBean_COLOR_32x32.gif");
18.     iconMono16 = loadImage("ChartBean_MONO_16x16.gif");
19.     iconMono32 = loadImage("ChartBean_MONO_32x32.gif");
20.
21.     try
22.     {
23.       PropertyDescriptor titlePositionDescriptor =
24.         new PropertyDescrip tor("titlePosition",ChartBean.class);
25.       titlePositionDescriptor.setPropertyEditorClass(TitlePositionEditor.class);
26.       PropertyDescriptor inverseDescriptor =
27.         new PropertyDescriptor("inverse", ChartBean.class);
28.       inverseDescriptor.setPropertyEditorClass(InverseEditor.class);
29.       PropertyDescriptor valuesDescriptor =
30.         new PropertyDescriptor("values", ChartBean.class);
31.       valuesDescriptor.setPropertyEditorClass(DoubleArrayEditor.class);
32.       propertyDescriptors = new PropertyDescriptor[] {
33.         new PropertyDescriptor("title", ChartBean.class),
34.         titlePositionDescriptor, valuesDescriptor,
35.         new PropertyDescriptor("graphColor", Chart Bean.class),
36.         inverseDescriptor };
37.     }
38.     catch (IntrospectionException e)
39.     {
40.       e.printStackTrace();
41.     }
42.   }
43.
44.   public PropertyDescriptor[] getPropertyDescriptors()
45.   {
46.     return propertyDescriptors;
47.   }
48.
49.   public Image getIcon(int iconType)
50.   {
51.     if (iconType == BeanInfo.ICON_COLOR_16x16) return iconColor16;
52.     else if (iconType == BeanInfo.ICON_COLOR_32x32) return iconColor32;
53.     else if (iconType == BeanInfo.ICON_MONO_16x16) return iconMono16;
54.     else if (iconType == BeanInfo.ICON_MONO_32x32) return iconMono32;
55.     else return null;
56.   }
```

```
57.    private PropertyDescriptor[] propertyDescriptors;
58.    private Image iconColor16;
59.    private Image iconColor32;
60.    private Image iconMono16;
61.    private Image iconMono32;
62. }
```

API java.beans.PropertyDescriptor 1.1

- `PropertyDescriptor(String name, Class<?> beanClass)`
  Создает объект `PropertyDescriptor`.
- `void setPropertyEditorClass(Class<?> editorClass)`
  Устанавливает класс редактора, подлежащий использованию с данным свойством.

API java.beans.BeanInfo 1.1

- `PropertyDescriptor[] getPropertyDescriptors()`
  Возвращает дескрипторы для каждого свойства, которое должно отображаться в инспекторе свойств данного компонента.

Создание редакторов свойств

Прежде чем углубляться в детали создания редакторов свойств, следует обратить внимание на то, что каждый такой редактор находится под управлением интегрированной среды разработки, а не bean-компонента. В частности, при отображении инспектора свойств, среда разработки выполняет для каждого свойства bean-компонента следующие действия.

1. Инициализирует редактор свойств.
2. Просит bean-компонент сообщить ей текущее значение свойства.
3. Затем просит редактор свойств отобразить это значение.

Редактор свойств должен иметь конструктор по умолчанию, а также обязательно реализовать интерфейс `PropertyEditor`. Обычно предпочтение отдается расширению такого полезного класса, как `PropertyEditorSupport`, который предоставляет версии методов, умеющие 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 `getAsText` и `setAsText`).
- В виде поле выбора (с помощью `getAsText`, `setAsText` и `getTags`).
- Графически за счет его прорисовки (с помощью `isPaintable`, `paintValue`, `supportsCustomEditor` и `getCustomEditor`).

Каждый из этих подходов более подробно рассматривается в следующих разделах.

Строковые редакторы свойств

Простые редакторы свойств работают с текстовыми строками за счет переопределения методов `setAsText()` и `getAsText()`. Например, у рассматриваемого здесь компонента `ChartBean` есть свойство, которое позволяет выбирать один из следую-

щих вариантов для размещения заголовка: слева, справа или по центру. Реализуются эти варианты с помощью такого перечисления:

```
public enum Position { LEFT, CENTER, RIGHT };
```

Конечно, отображаться так как есть, т.е. в виде строк с заглавными буквами — `LEFT`, `CENTER`, `RIGHT` — они, во избежание лишних проблем с пользовательским интерфейсом, не должны. Поэтому и определяется редактор свойств, метод `getAsText()` которого подбирает для отображения строку более приглядного с точки зрения разработчика вида:

```
class TitlePositionEditor extends PropertyEditorSupport
{
  public String getAsText()
  {
    int index = ((ChartBean.Position) getValue()).ordinal();
    return tags[index];
  }
  . . .
  private String[] tags = { "Left", "Center", "Right" };
}
```

В идеале эти строки должны отображаться на текущем локальном языке, а не обязательно на английском, но мы оставляем это в качестве упражнения для самостоятельной работы.

Естественно, теперь также еще необходимо добавить и 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 мы просто предоставим эти два метода, инспектор свойств будет отображать текстовое поле. Оно будет инициализироваться вызовом метода `getAsText`, а по завершении выполнения в нем редактирования будет вызываться метод `setAsText`. Для свойства `titlePosition`, конечно, такой вариант является далеко не самым подходящим. В его случае будет лучше сделать так, чтобы все действительные значения отображались не в текстовом поле, а в комбинированном окне, как показано на рис. 8.11.

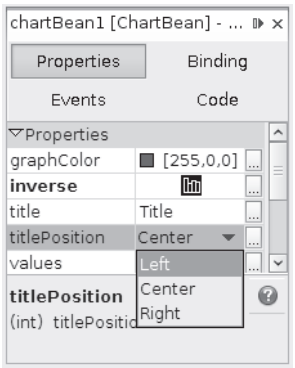


Рис. 8.11. Специальный редактор свойств в действии

Класс PropertyEditorSupport предоставляет простой механизм для указания того, что должно использоваться комбинированное окно. Все, что требуется — это всего лишь написать метод getTags(), возвращающий массив соответствующих строк:

```
public String[] getTags() { return tags; }
```

По умолчанию метод getTags() возвращает значение null, указывающее на то, что текстовое поле является подходящим для редактирования значения свойства.

В случае предоставления метода getTags() методы getAsText() и setAsText() предоставлять все равно нужно. Метод getTags() просто перечисляет строки, которые должны предлагаться пользователю, а методы getAsText() и setAsText() отвечают за преобразование этих строк в тип данных свойства (который может представлять собой строку, целое число, перечисление или еще какой-нибудь совершенно другой тип) и обратно.

И, наконец, еще редакторы свойств должны реализовать метод getJavaInitialization String. С помощью этого метода интегрированной среде разработки можно передавать Java-строку, устанавливающую для свойства его текущее значение, чтобы та использовала ее для автоматической генерации кода. Для рассматриваемого здесь редактора TitlePositionEditor этот метод должен выглядеть следующим образом:

```
public String getJavaInitializationString()
{
  return ChartBean.Position.class.getName().replace('$', '.') + "." + getValue();
}
```

Он будет возвращать примерно такую строку: com.horstmann.corejava.ChartBean.Position.LEFT.

Испробуйте его в среде NetBeans. В случае редактирования свойства titlePosition среда NetBeans должна вставить следующую строку кода:

```
chartBean1.setTitlePosition(com.horstmann.corejava.ChartBean.Position.LEFT);
```

В нашей ситуации код выглядит немного громоздко, поскольку ChartBean.Position.class.getName() возвращает строку com.horstmann.corejava.ChartBean$Position. Далее остается только заменить символ $ символом точки и добавить результат вызова toString к значению перечисления.



При наличии у свойства специального редактора, не реализующего метод getJavaInitialization String, среда NetBeans не будет знать, как генерировать код, и потому будет создавать set-метод с параметром ???.

В листинге 8.4 приведен полный код редактора TitlePositionEditor.

Листинг 8.4. Содержимое файла TitlePositionEditor.java

```
1.  package com.horstmann.corejava;
2.
3.  import java.beans.*;
4.  import java.util.*;
5.
6.  /**
7.   * Специальный редактор для свойства titlePosition компонента
8.   * ChartBean, который позволяет пользователю выбирать
9.   * между значениями Left, Center и Right.
10.  * @version 1.20 2007-12-14
11.  * @author Cay Horstmann
12.  */
```

```
13.  public class TitlePositionEditor extends PropertyEditorSupport
14.  {
15.    public String[] getTags()
16.    {
17.      return tags;
18.    }
19.
20.    public String getJavaInitializationString()
21.    {
22.      return ChartBean.Position.class.getName().replace('$', '.') + "." + getValue();
23.    }
24.
25.    public String getAsText()
26.    {
27.      int index = ((ChartBean.Position) getValue()).ordinal();
28.      return tags[index];
29.    }
30.
31.    public void setAsText(String s)
32.    {
33.      int index = Arrays.asList(tags).indexOf(s);
34.      if (index >= 0) setValue(ChartBean.Position.values()[index]);
35.    }
36.
37.    private String[] tags = { "Left", "Center", "Right" };
38.  }
```

Визуальные редакторы свойств

Сложные свойства не могут представляться в текстовом виде. Вместо этого они имеют графическое представление и отображаются в инспекторе свойств в виде небольшой области на том же месте, где в противном случае могло бы находиться текстовое поле или комбинированное окно. При выполнении пользователем щелчка в этой области появляется диалоговое окно специального редактора, подобному тому, которое показано на рис. 8.12.

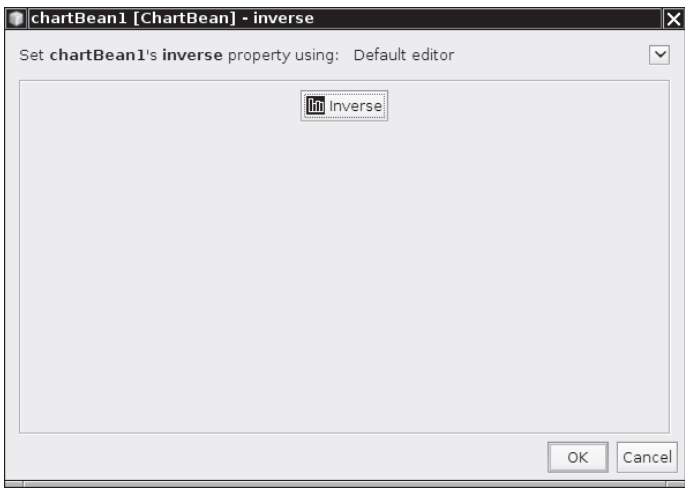


Рис. 8.12. Диалоговое окно специального визуального редактора свойств

В диалоговом окне содержится компонент для редактирования значения свойства, предоставляемый самим редактором свойств, а также различные кнопки, предоставляемые интегрированной средой разработки. В рассматриваемом здесь примере демонстрируется довольно простой визуальный редактор, содержащий единственную кнопку. Однако в составе загружаемого кода для этой книги можно найти пример и более сложного редактора, позволяющего редактировать значения диаграммы.

При создании визуального редактора первым делом инспектору свойств сообщается о том, что значение будет прорисовываться, а не отображаться путем использования строки.

Далее метод `getAsText()` интерфейса `PropertyEditor` переопределяется так, чтобы он возвращал значение `null`, а метод `isPaintable()` — так, чтобы он возвращал значение `true`.

После этого реализуется метод `paintValue()`. Этот метод получает контекст `Graphics` и координаты прямоугольника, внутри которого может выполняться прорисовка. Обратите внимание на то, что данный прямоугольник обычно имеет довольно маленькие размеры, так что выполнять в нем какую-то очень сложную прорисовку невозможно. В нашем примере в нем просто прорисовывается одна из двух возможных пиктограмм (увидеть которые можно 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);
}
```

Конечно, это графическое представление не предназначено для редактирования. Пользователь должен просто щелкать на его поверхности для отображения диалогового окна специального редактора.

То, что будет использоваться специальный редактор, указывается за счет переопределения метода `supportCustomEditor()` интерфейса `PropertyEditor` таким образом, чтобы он возвращал значение `true`.

Далее метод `getCustomEditor()` интерфейса `PropertyEditor` просто создает и возвращает объект класса этого специального редактора.

В листинге 8.5 приведен код редактора `InverseEditor`, отображающий текущее значение свойства в инспекторе свойств, а в листинге 8.6 — код панели этого специального редактора, позволяющей редактировать это значение.

Листинг 8.5. Содержимое файла `InverseEditor.java`

```
1.  package com.horstmann.corejava;
2.
3.  import java.awt.*;
4.  import java.beans.*;
5.  import javax.swing.*;
6.
7.  /**
8.   * Редактор для свойства inverse компонента ChartBean.
9.   * Это свойство изменяет цвет столбцов и фона диаграммы.
10.  * @version 1.30 2007-10-03
11.  * @author Cay Horstmann
12.  */
13. public class InverseEditor extends PropertyEditorSupport
14. {
```

```
15.   public Component getCustomEditor()
16.   {
17.     return new InverseEditorPanel(this);
18.   }
19.
20.   public boolean supportsCustomEditor()
21.   {
22.     return true;
23.   }
24.
25.   public boolean isPaintable()
26.   {
27.     return true;
28.   }
29.
30.   public String getAsText()
31.   {
32.      return null;
33.   }
34.
35.   public String getJavaInitializationString()
36.   {
37.     return "" + getValue();
38.   }
39.
40.   public void paintValue(Graphics g, Rectangle bounds)
41.   {
42.     ImageIcon icon = (Boolean) getValue() ? inverseIcon : normalIcon;
43.     int x = bounds.x + (bounds.width - icon.getIconWidth()) / 2;
44.     int y = bounds.y + (bounds.height - icon.getIconHeight()) / 2;
45.     g.drawImage(icon.getImage(), x, y, null);
46.   }
47.
48.   private ImageIcon inverseIcon = new ImageIcon(getClass().getResource(
49.      "ChartBean_INVERSE_16x16.gif"));
50.   private ImageIcon normalIcon =
51.        new ImageIcon(getClass().getResource("ChartBean_MONO_16x16.gif"));
52. }
```

Листинг 8.6. Содержимое файла `InverseEditorPanel.java`

```
1.  package com.horstmann.corejava;
2.
3.  import java.awt.event.*;
4.  import java.beans.*;
5.  import javax.swing.*;
6.
7.  /**
8.   * Панель для установки свойства inverse.
9.   * Содержит переключатели обычного и инверсного цвета.
10.  * @version 1.30 2007-10-03
11.  * @author Cay Horstmann
12.  */
13. public class InverseEditorPanel extends JPanel
14. {
15.   public InverseEditorPanel(PropertyEditorSupport ed)
16.   {
```

```
17.     editor = ed;
18.     button = new JButton();
19.     updateButton();
20.     button.addActionListener(new ActionListener()
21.       {
22.         public void actionPerformed(ActionEvent event)
23.         {
24.           editor.setValue(!(Boolean) editor.getValue());
25.           updateButton();
26.         }
27.       });
28.     add(button);
29.   }
30.
31.   private void updateButton()
32.   {
33.     if ((Boolean) editor.getValue())
34.     {
35.       button.setIcon(inverseIcon);
36.       button.setText("Inverse");
37.     }
38.     else
39.     {
40.       button.setIcon(normalIcon);
41.       button.setText("Normal");
42.     }
43.   }
44.
45.   private JButton button;
46.   private PropertyEditorSupport editor;
47.   private ImageIcon inverseIcon = new ImageIcon(getClass().getResource(
48.     "ChartBean_INVERSE_16x16.gif"));
49.   private ImageIcon normalIcon =
50.       new ImageIcon(getClass().getResource("ChartBean_MONO_16x16.gif"));
51. }
```

API java.beans.PropertyEditor 1.1

- `Object getValue()`

  Возвращает текущее значение свойства. Базовые типы упаковываются в оболочки объектов.

- `void setValue(Object newValue)`

  Устанавливает для свойства новое значение. Базовые типы должны помещаться в оболочки объектов.

  Параметры: `newValue` Новое значение объекта; это должен быть новый, только что созданный объект, которым может владеть данное свойство

- `String getAsText()`

  Этот метод должен переопределяться так, чтобы он возвращал строковое представление текущего значения свойства. По умолчанию он возвращает значение `null`, указывающее на то, что данное свойство не может представляться в виде строки.

- `void setAsText(String text)`

  Этот метод должен переопределяться так, чтобы он устанавливал для свойства новое значение, получаемое путем выполнения синтаксического анализа текста. Он может выдавать исключение `IllegalArgumentException` в случае, если текст не предоставляет действительного значения или если данное свойство не может быть представлено в виде строки.

- `String[] getTags()`

  Этот метод должен переопределяться так, чтобы он возвращал массив всех возможных строковых представления значений свойства, дабы они могли отображаться в комбинированном окне вариантов. По умолчанию он возвращает значение `null`, указывающее на то, что конечного множества строковых значений не существует.

- `boolean isPaintable()`

  Этот метод должен переопределяться так, чтобы он возвращал значение `true` в случае, если класс использует для отображения свойства метод `paintValue()`.

- `void paintValue(Graphics g, Rectangle box)`

  Этот метод должен переопределяться так, чтобы он отображал значение путем его прорисовывания в графическом контексте внутри специально отведенной для этого области в компоненте, используемом для инспектора свойств.

- `boolean supportsCustomEditor()`

  Этот метод должен переопределяться так, чтобы он возвращал значение `true`, если у редактора свойств имеется какой-то специальный редактор.

- `Component getCustomEditor()`

  Этот метод должен переопределяться так, чтобы он возвращал компонент, реализующий специальный пользовательский интерфейс для редактирования значения свойства.

- `String getJavaInitializationString()`

  Этот метод должен переопределяться так, чтобы он возвращал строку Java-кода, пригодную для генерации кода, инициализирующего значение свойства, например, `"0"`, `"new Color(64, 64, 64)"`.

Настройщики

Редактор свойств позволяет пользователю устанавливать только по одному свойству за раз. Особенно в тех случаях, когда определенные свойства bean-компонента связаны между собой, лучше предоставлять пользователям возможность редактировать одновременно сразу несколько свойств. Для обеспечения такой возможности вместо (или в дополнение) к редакторам свойств может предоставляться *настройщик* (customizer).

Более того, некоторые bean-компоненты могут иметь возможности, которые просто невозможно предоставлять в виде свойств и, следовательно, редактировать посредством инспектора свойств. В случае таких компонентов настройщик играет очень важную роль.

В рассматриваемом в данном разделе примере демонстрируется настройщик для компонента `ChartBean`. Этот настройщик позволяет устанавливать значения сразу для нескольких свойств компонента в одном диалоговом окне, как можно увидеть на рис. 8.13.

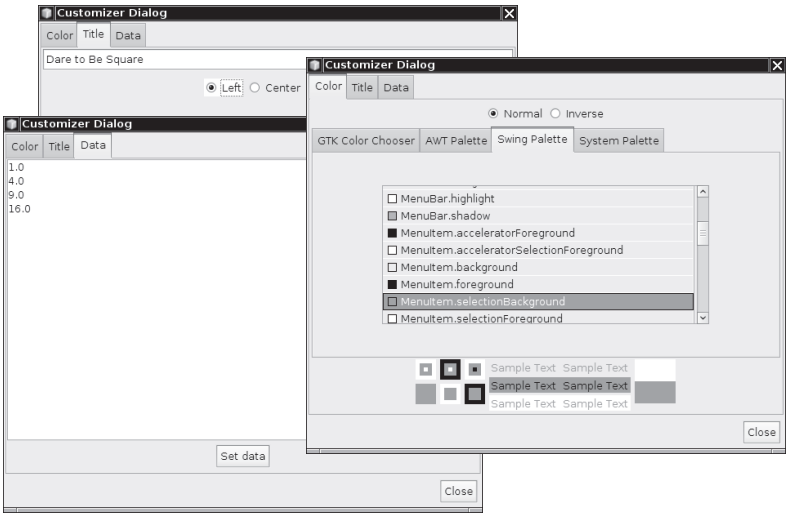


Рис. 8.13. Настройщик компонента `ChartBean`

Для создания настройщика в компоненте JavaBeans нужно создать класс `BeanInfo` и переопределить его метод `getBeanDescriptor()`, как показано ниже:

```
public ChartBean2BeanInfo extends SimpleBeanInfo
{
   public BeanDescriptor getBeanDescriptor()
   {
     return beanDescriptor;
   }
   . . .
    private BeanDescriptor beanDescriptor
      = new BeanDescriptor(ChartBean2.class, ChartBean2Customizer.class);
}
```

Обратите внимание на то, что при создании класса настройщика обязательно следовать каким-либо соглашениям об именовании не требуется. (Хотя все-таки принято именовать такие классы по схеме *ИмяКомпонента*`Customizer`.)

Реализация класса настройщика рассматривается в следующем разделе.

API java.beans.BeanInfo 1.1

- `BeanDescriptor getBeanDescriptor()`

  Возвращает объект `BeanDescriptor`, который описывает возможности компонента JavaBeans.

API java.beans.BeanDescriptor 1.1

- `BeanDescriptor(Class<?> beanClass, Class<?> customizerClass)`

  Создает объект `BeanDescriptor` для компонента, обладающего настройщиком.

Параметры:	`beanClass`	Объект `Class` для компонента
	`customizerClass`	Объект `Class` для настройщика компонента

Создание класса настройщика

Каждый создаваемый класс настройщика должен иметь используемый по умолчанию конструктор, расширять класс `Component` и реализовать интерфейс `Customizer`. Интерфейс `Customizer` предусматривает всего лишь три следующих метода.

- Метод `setObject()`, который принимает в качестве параметра подлежащий настройке компонент JavaBeans.
- Методы `addPropertyChangeListener()` и `removePropertyChangeListener()`, которые отвечают за управление набором обработчиков событий изменения свойства в настройщике.

Рекомендуется обновлять графический интерфейс компонента при изменении пользователем значения любого свойства, а не только по окончании процесса настройки.

В отличие от редакторов свойств, настройщики не отображаются автоматически. В интегрированной среде разработки NetBeans для вызова настройщика нужно щелкать правой кнопкой мыши на компоненте и выбирать пункт меню Customize (Настроить). В ответ среда разработки вызывает метод `setObject()` настройщика. Обратите внимание, что создание настройщика происходит до его фактического связывания с экземпляром компонента JavaBeans. Поэтому предполагать наличие какой-либо информации о состоянии компонента в конструкторе нельзя.

Поскольку настройщики обычно предоставляют пользователю возможность работы с несколькими параметрами, зачастую для них наиболее удобно использовать панель с несколькими вкладками. Именно такой подход как раз и применяется в демонстрируемом здесь примере, для чего класс настройщика расширяет класс `JTabbedPane`.

Настройщик размещает информацию на трех вкладках, обеспечивающих следующие возможности:

- выбор цвета и режима отображения;
- выбор заголовка и его расположения;
- выбор отображаемых в диаграмме данных.

Конечно, для создания такого пользовательского интерфейса потребуется выполнить очень большой объем рутинной работы. В нашем примере только конструктор содержит более 100 строк кода. Однако для его создания понадобятся только самые базовые навыки программирования компонентов Swing, а потому мы не будем рассматривать несущественные подробности этого процесса.

При создании настройщика стоит помнить об одном полезном приеме. В настройщике часто требуется редактировать значения свойств. Так вот, вместо того, чтобы реализовывать новый интерфейс для установки значения свойства какого-то конкретного класса, можно просто находить уже существующий редактор свойств и добавлять его в свой пользовательский интерфейс! Например, в нашем настройщике `ChartBean2` требуется устанавливать цвет диаграммы. Поскольку мы знаем, что в NetBeans имеется отличный редактор свойств цветов, мы просто устанавливаем его местонахождение следующим образом:

```
PropertyEditor colorEditor = PropertyEditorManager.findEditor(Color.Class);
Component colorEditorComponent = colorEditor.getCustomEditor();
```

После размещения компонентов мы инициализируем их значения с помощью метода `setObject()`. Этот метод вызывается при отображении настройщика. В роли его параметра выступает подлежащий настройке компонент JavaBeans. Для продолже-

ния мы сохраняем ссылку на этот компонент, поскольку она понадобится позже для уведомления компонента об изменении его свойств. Затем мы инициализируем каждый компонент пользовательского интерфейса. Выполняющий эту инициализацию фрагмент кода в методе `setObject()` настройщика компонента `ChartBean2` выглядит следующим образом:

```
public void setObject(Object obj)
{
   bean = (ChartBean2)obj;
   titleField.setText(bean.getTitle());
   colorEditor.setValue(bean.getGraphColor());
   ...
}
```

Напоследок мы присоединяем обработчики событий для отслеживания действий пользователя. Всякий раз, когда пользователь изменяет значение компонента, компонент генерирует событие, которое наш настройщик должен обязательно обрабатывать. В частности, он должен делать следующее: обновлять значение соответствующего свойства в компоненте и дополнительно генерировать событие `PropertyChangeEvent` для того, чтобы другие обработчики (вроде инспектора свойств) тоже могли выполнить соответствующее обновление. Давайте проследим этот процесс на примере нескольких элементов пользовательского интерфейса в настройщике компонента `ChartBean2`.

В случае ввода пользователем нового заголовка мы хотим, чтобы происходило обновление свойства заголовка. Поэтому мы присоединяем к тому текстовому полю, в котором пользователь будет вводить заголовок, обработчик `DocumentListener`.

```
titleField.getDocument().addDocumentListener(new
   DocumentListener()
   {
      public void changeUpdate(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 bean-компонент для выполнения обновления значения его соответствующего свойства и затем генерирует событие изменения свойства. (Такое обновление необходимо только для несвязанных свойств.) Код этого метода выглядит следующим образом:

```
public void setTitle(String newValue)
{
   if (bean == null) return;
   String oldValue = bean.getTitle();
   bean.setTitle(newValue);
   firePropertyChange("title", oldValue, newValue);
}
```

В случае изменения значения цвета в редакторе свойства цвета мы хотим, чтобы в компоненте происходило обновление цвета диаграммы. Поэтому для отслеживания изменений цвета мы присоединяем к этому редактору слушателя. Этот редактор сам тоже рассылает события, уведомляющие об изменении значения свойства:

```
colorEditor.addPropertyChangeListener(new
   PropertyChangeListener()
   {
      public void propertyChange(PropertyChangeEvent event)
      {
         setGraphColor((Color) colorEditor.getValue());
      }
   });
```

В листинге 8.7 приведен полный код настройщика компонента `ChartBean2`.

Листинг 8.7. Содержимое файла `ChartBean2Customizer.java`

```
1.  package com.horstmann.corejava;
2.  import java.awt.*;
3.  import java.awt.event.*;
4.  import java.beans.*;
5.  import java.util.*;
6.  import javax.swing.*;
7.  import javax.swing.event.*;
8.
9.  /**
10.  * Настройщик для bean-компонента диаграмм, который позволяет
11.  * пользователю редактировать все свойства диаграммы внутри
12.  * единственного диалогового окна с вкладками.
13.  * @version 1.12 2007-10-03
14.  * @author Cay Horstmann
15.  */
16. public class ChartBean2Customizer extends JTabbedPane implements Customizer
17. {
18.   public ChartBean2Customizer()
19.   {
20.     data = new JTextArea();
21.     JPanel dataPane = new JPanel();
22.     dataPane.setLayout(new BorderLayout());
23.     dataPane.add(new JScrollPane(data), BorderLayout.CENTER);
24.     JButton dataButton = new JButton("Set data");
25.     dataButton.addActionListener(new ActionListener()
26.       {
27.         public void actionPerformed(ActionEvent event)
28.         {
29.           setData(data.getText());
30.         }
31.       });
32.     JPanel panel = new JPanel();
33.     panel.add(dataButton);
34.     dataPane.add(panel, BorderLayout.SOUTH);
35.
36.     JPanel colorPane = new JPanel();
37.     colorPane.setLayout(new BorderLayout());
38.
39.     normal = new JRadioButton("Normal", true);
40.     inverse = new JRadioButton("Inverse", false);
```

```
41.     panel = new JPanel();
42.     panel.add(normal);
43.     panel.add(inverse);
44.     ButtonGroup group = new ButtonGroup();
45.     group.add(normal);
46.     group.add(inverse);
47.     normal.addActionListener(new ActionListener()
48.       {
49.         public void actionPerformed(ActionEvent event)
50.         {
51.           setInverse(false);
52.         }
53.       });
54.
55.     inverse.addActionListener(new ActionListener()
56.      {
57.         public void actionPerformed(ActionEvent event)
58.         {
59.           setInverse(true);
60.         }
61.       });
62.
63.     colorEditor = PropertyEditorManager.findEditor(Color.class);
64.     colorEditor.addPropertyChangeListener(new PropertyChangeListener()
65.       {
66.         public void propertyChange(PropertyChangeEvent event)
67.         {
68.            setGraphColor((Color) colorEditor.getValue());
69.         }
70.      });
71.
72.     colorPane.add(panel, BorderLayout.NORTH);
73.     colorPane.add(colorEditor.getCustomEditor(), BorderLayout.CENTER);
74.
75.     JPanel titlePane = new JPanel();
76.     titlePane.setLayout(new BorderLayout());
77.
78.     group = new ButtonGroup();
79.     position = new JRadioButton[3];
80.     position[0] = new JRadioButton("Left");
81.     position[1] = new JRadioButton("Center");
82.     position[2] = new JRadioButton("Right");
83.
84.     panel = new JPanel();
85.     for (int i = 0; i < position.length; i++)
86.     {
87.       final ChartBean2.Position pos = ChartBean2.Position.values()[i];
88.       panel.add(position[i]);
89.       group.add(position[i]);
90.       position[i].addActionListener(new ActionListener()
91.       {
92.         public void actionPerformed(ActionEvent event)
93.         {
94.            setTitlePosition(pos);
95.          }
96.     });
97.   }
```

```
98.   titleField = new JTextField();
99.   titleField.getDocument().addDocumentListener(new DocumentListener()
100.    {
101.     public void changedUpdate(DocumentEvent evt)
102.     {
103.       setTitle(titleField.getText());
104.      }
105.
106.     public void insertUpdate(DocumentEvent evt)
107.     {
108.       setTitle(titleField.getText());
109.     }
110.
111.     public void removeUpdate(DocumentEvent evt)
112.     {
113.      setTitle(titleField.getText());
114.     }
115.   });
116.
117.  titlePane.add(titleField, BorderLayout.NORTH);
118.  JPanel panel2 = new JPanel();
119.  panel2.add(panel);
120.  titlePane.add(panel2, BorderLayout.CENTER);
121.  addTab("Color", colorPane);
122.  addTab("Title", titlePane);
123.  addTab("Data", dataPane);
124. }
125. /**
126.  * Установка позиции заголовка диаграммы.
127.  * @param i Новая позиция заголовка (ChartBean2.LEFT,
128.           ChartBean2.CENTER или ChartBean2.RIGHT)
129.  */
130.  public void setTitlePosition(ChartBean2.Position pos)
131.  {
132.    if (bean == null) return;
133.    ChartBean2.Position oldValue = bean.getTitlePosition();
134.    bean.setTitlePosition(pos);
135.    firePropertyChange("titlePosition", oldValue, pos);
136.  }
137.
138.  /**
139.   * Установка режима отображения
140.   * (нормальный или инвертированный).
141.   * @param b Равен true для инвертированного изображения
142.   */
143.  public void setInverse(boolean b)
144.  {
145.    if (bean == null) return;
146.    boolean oldValue = bean.isInverse();
147.    bean.setInverse(b);
148.    firePropertyChange("inverse", oldValue, b);
149.  }
150.
151.  /**
152.   * Установка значений, подлежащих отображению в диаграмме.
153.   * @param newValue Новый массив значений
154.   */
```

```
155.  public void setValues(double[] newValue)
156.  {
157.    if (bean == null) return;
158.    double[] oldValue = bean.getValues();
159.    bean.setValues(newValue);
160.    firePropertyChange("values", oldValue, newValue);
161.  }
162.  /**
163.   * Установка цвета диаграммы.
164.   * @param newValue Новый цвет
165.   */
166.  public void setGraphColor(Color newValue)
167.  {
168.    if (bean == null) return;
169.    Color oldValue = bean.getGraphColor();
170.    bean.setGraphColor(newValue);
171.    firePropertyChange("graphColor", oldValue, newValue);
172.  }
173.
174.  public void setObject(Object obj)
175.  {
176.    bean = (ChartBean2) obj;
177.
178.    data.setText("");
179.    for (double value : bean.getValues())
180.      data.append(value + "\n");
181.
182.    normal.setSelected(!bean.isInverse());
183.    inverse.setSelected(bean.isInverse());
184.
185.    titleField.setText(bean.getTitle());
186.
187.    for (int i = 0; i < position.length; i++)
188.      position[i].setSelected(i == bean.getTitlePosition().ordinal());
189.
190.    colorEditor.setValue(bean.getGraphColor());
191.  }
192.
193.  private ChartBean2 bean;
194.  private PropertyEditor colorEditor;
195.
196.  private JTextArea data;
197.  private JRadioButton normal;
198.  private JRadioButton inverse;
199.  private JRadioButton[] position;
200.  private JTextField titleField;
201. }
```

API java.beans.Customizer 1.1

- `void setObject(Object bean)`
  Определяет настраиваемый компонент JavaBeans.

Постоянные компоненты JavaBeans

Для обеспечения постоянства компоненты JavaBeans сохраняются в потоке и восстанавливаются позже, возможно, под управлением другой виртуальной машины. На первый взгляд может показаться, что средства реализации постоянства JavaBeans не отличаются от средств сериализации объектов (см. главу 12 тома I). Однако между этими механизмами есть существенное различие. Постоянные компоненты JavaBeans *допускают долговременное хранение.*

При сериализации объектов переменные экземпляра записываются в поток. Если реализация класса изменится, могут измениться и его поля. В результате обычное чтение из файла, содержащего сериализированный объект старой версии, может дать при работе с новой версией класса некорректный результат. В некоторых случаях удается учесть различия между версиями и реализовать отображение старых данных в новые переменные, однако этот процесс требует слишком много времени и усилий и не всегда приводит к успеху. Таким образом, сериализованные объекты непригодны для длительного хранения. Поэтому в документации для всех компонентов Swing содержится предупреждение о том, что экземпляры данного класса могут оказаться несовместимыми с последующими реализациями, и хранение сериализованных объектов допустимо лишь на время, необходимое для передачи их от одного приложения другому.

Решением данной проблемы являются постоянные компоненты. Первоначально механизм обеспечения постоянства предназначался для разработки графического пользовательского интерфейса. Инструментальные средства сохраняли в файле результаты действий с мышью над фреймами, панелями, кнопками и другими компонентами Swing. Выполняющаяся программа открывала этот файл и считывала его содержимое. Такой подход позволял обойтись без обширных исходных кодов, предназначенных для размещения компонентов Swing и ввода данных. К сожалению, он не был принят для широкого применения.



Механизм постоянства использовался в экспериментальной графической среде разработки Bean Builder. Этот продукт доступен по адресу: `http://bean-builder.dev.java.net`.

Идея, лежащая в основе постоянных компонентов JavaBeans, проста. Предположим, вам требуется сохранить объект `JFrame` и восстановить его позже. Если вы обратите внимание на исходные коды класса `JFrame` и его суперклассов, то увидите десятки переменных экземпляра. Для сериализации фрейма необходимо сохранить их текущие значения. Однако обратите внимание на процесс создания фрейма:

```
JFrame frame = new JFrame();
frame.setTitle("My Application");
frame.setVisible(true);
```

Конструктор по умолчанию инициализирует все переменные экземпляра и устанавливает несколько свойств. При сохранении объекта `frame` средства обеспечения постоянства JavaBeans запишут соответствующие операторы в формате XML:

```
<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 `JFrame`, и устанавливаются значения свойств `title` и `visible`. При этом не имеет значения, изменилась ли внутренняя структура `JFrame`. Таким образом, вы можете восстановить любой объект, устанавливая его свойства.

Обратите внимание, что сохраняться должны только те свойства, значения которых отличаются от устанавливаемых по умолчанию. Объект `XMLEncoder` создает `JFrame` и сравнивает его свойства с фреймом, подлежащим архивированию. Таким образом, свойства, имеющие значения по умолчанию, не записываются. Этот механизм называется *устранением избыточности* (redundancy elimination). В результате размер архива становится значительно меньше, чем при сериализации. (Особенно это заметно при сериализации компонентов Swing. Их состояние определяется множеством переменных экземпляра, значения которых почти никогда не отличаются от значений по умолчанию.)

Описанный подход не лишен недостатков. Рассмотрим, например, вызов метода, который нельзя считать установкой значения свойства:

```
frame.setSize(600, 400);
```

Однако `XMLEncoder` справляется с этой проблемой. Он записывает в файл 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, необходимо создать экземпляр класса `XMLEncoder`:

```
XMLEncoder out = new XMLEncoder(new FileOutputStream(. . .));
out.writeObject(frame);
out.close();
```

Аналогично, для чтения сохраненного объекта используется класс `XMLDecoder`:

```
XMLDecoder in = new XMLDecoder(new FileInputStream(. . .));
JFrame newFrame = (JFrame) in.readObject();
in.close();
```

Программа, код которой приведен в листинге 8.8, демонстрирует сохранение и восстановление фрейма (рис. 8.14), причем инициатором процессов записи и чтения состояния является *сам* фрейм.

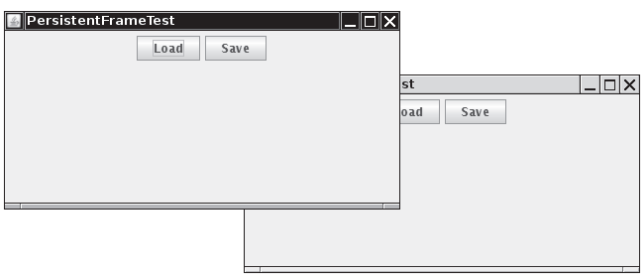


Рис. 8.14. Программа `PersistentFrameTest` в действии

После запуска этой программы, сначала щелкните на кнопке Save (Сохранить) и сохраните фрейм в каком-нибудь файле. Затем переместите исходный фрейм в другое место и щелкните на кнопке Load (Загрузить), чтобы увидеть, как на том месте, где прежде находился исходный фрейм, появится новый. Загляните в XML-файл, который сгенерирует эта программа.

Если вы внимательно посмотрите на выходные XML-данные этой программы, то увидите, что объект `XMLEncoder` при сохранении фрейма выполняет удивительно много работы. В частности, он генерирует операторы, которые выполняют следующие действия:

- устанавливают различные свойств фрейма (вроде `size`, `layout`, `defaultCloseOperation`, `title` и т.д.);
- добавляют в состав фрейма различные кнопки;
- присоединяют к кнопкам обработчики событий.

Здесь нам пришлось создавать обработчики событий с помощью класса `EventHandler`. Архивировать произвольные внутренние классы объект `XMLEncoder` не умеет, а вот как обрабатывать объекты `EventHandler` ему известно.

Листинг 8.8. Содержимое файла `PersistentFrameTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.beans.*;
4.  import java.io.*;
5.  import javax.swing.*;
6.
7.  /**
8.   * Данная программа демонстрирует использование XMLEncoder
9.   * и XMLDecoder для сохранения и восстановления фрейма.
10.  * @version 1.01 2007-10-03
11.  * @author Cay Horstmann
12.  */
13. public class PersistentFrameTest
14. {
15.    public static void main(String[] args)
16.    {
17.       chooser = new JFileChooser();
18.       chooser.setCurrentDirectory(new File("."));
19.       PersistentFrameTest test = new PersistentFrameTest();
20.       test.init();
21.    }
22.
23.    public void init()
24.    {
25.       frame = new JFrame();
26.       frame.setLayout(new FlowLayout());
27.       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28.       frame.setTitle("PersistentFrameTest");
29.       frame.setSize(400, 200);
30.
31.       JButton loadButton = new JButton("Load");
32.       frame.add(loadButton);
33.       loadButton.addActionListener(EventHandler.create(
34.          ActionListener.class, this, "load"));
35.
```

```
36.       JButton saveButton = new JButton("Save");
37.       frame.add(saveButton);
38.       saveButton.addActionListener(EventHandler.create(
39.         ActionListener.class, this, "save"));
40.
41.       frame.setVisible(true);
42.     }
43.
44.     public void load()
45.     {
46.       // Отображение диалогового окна выбора файлов.
47.       int r = chooser.showOpenDialog(null);
48.
49.       // Если файл выбран, открыть его.
50.       if(r == JFileChooser.APPROVE_OPTION)
51.       {
52.         try
53.         {
54.           File file = chooser.getSelectedFile();
55.           XMLDecoder decoder = new
56.             XMLDecoder(new FileInputStream(file));
57.           JFrame newFrame = (JFrame) decoder.readObject();
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.
86.    private static JFileChooser chooser;
87.    private JFrame frame;
88.  }
```

Использование средств обеспечения постоянства JavaBeans для работы с произвольными данными

Область применения средств обеспечения постоянства JavaBeans не ограничивается сохранением компонентов Swing. Данный механизм можно использовать для записи в поток любого набора объектов, удовлетворяющих простым условиям. Читая следующие разделы, вы научитесь использовать средства обеспечения постоянства JavaBeans для долговременного хранения данных.

Объект `XMLEncoder` имеет встроенную поддержку для следующих типов:

- `null`;
- всех примитивных типов и их оболочек;
- `String`;
- массивов;
- коллекций и карт (map);
- типов рефлексии `Class`, `Field`, `Method` и `Proxy`;
- AWT-типов `Color`, `Cursor`, `Dimension`, `Font`, `Insets`, `Point`, `Rectangle` и `ImageIcon`;
- компонентов AWT и Swing, границ, диспетчеров компоновки и моделей;
- обработчиков событий.

Использование делегатов при создании объектов

Применение средств обеспечения постоянства JavaBeans не составляет труда в случае, если состояние объектов изменяется путем установки их свойств. Однако поведение реальных программ не всегда вписывается в эти рамки. Возьмем, например, класс `Employee`, код которого приводился в главе 4 первого тома. Очевидно, что его нельзя считать полноценным объектом bean. У него нет ни конструктора по умолчанию, ни методов `setName()`, `setSalary()`, `setHireDay()`. Для устранения этой проблемы можно определить для него так называемого *делегата обеспечения постоянства* (persistence delegate). Такой делегат будет отвечать за генерацию XML-кода для объекта `Employee`.

Делегат класса `Employee` должен переопределять метод `instantiate()` и генерировать *выражение*, конструирующее объект:

```
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)
           });
       }
     };
```

Данный фрагмент кода означает: "Для восстановления oldInstance вызовите метод new (т.е. конструктор) объекта Employee.class и предоставьте данные параметры". Имя параметра oldInstance может немного вводить в заблуждение — на самом деле под ним подразумевается просто сохраняемый экземпляр класса.

Установить делегата можно двумя способами — либо путем его сопоставления с конкретным XMLWriter:

```
out.setPersistenceDelegate(Employee.class, delegate);
```

либо путем установки атрибута persistenceDelegate для *дескриптора bean-компонента* (bean descriptor) класса BeanInfo:

```
BeanInfo info = Introspector.getBeanInfo(GregorianCalendar.class);
info.getBeanDescriptor().setValue("persistenceDelegate", delegate);
```

После инсталляции делегата можно уже переходить и непосредственно к сохранению объектов Employee. Например, приведенные ниже операторы:

```
Object myData = new Employee("Harry Hacker", 50000, 1989, 10, 1);
out.writeObject(myData);
```

будут генерировать следующие выходные данные:

```
<object class="Employee">
   <string>Harry Hacker</string>
   <double>50000.0</double>
   <int>1989</int>
   <int>10</int>
   <int>1</int>
</object>
```



Настройке подлежит только процесс *кодирования*. Специальные методы декодирования не используются. Для декодирования достаточно выполнить операторы из XML-файла.

Создание объекта из свойств

В некоторых случаях удается упростить работу по созданию делегата. Если все параметры конструктора можно получить путем обращения к свойствам oldInstance, создавать метод instantiate() нет необходимости. Достаточно сформировать объект DefaultPersistenceDelegate и предоставить ему имена свойств.

Например, приведенный ниже оператор устанавливает делегата для класса Rectangle2D.Double:

```
out.setPersistenceDelegate(Rectangle2D.Double.class,
   new DefaultPersistenceDelegate(new String[] { "x", "y",
   "width", "height" }));
```

Он означает следующее: "Для кодирования объекта Rectangle2D.Double нужно получить его свойства x, y, width и height и передать эти четыре значения конструктору". В результате будет сгенерирован показанный ниже элемент:

```
<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 делать даже еще лучше, а именно — снабжать конструктор аннотацией @ConstructorProperties. Например, если бы у класса Employee был

конструктор с тремя параметрами (в роли которых выступали, скажем, имя сотрудника, его заработная плата и день приема на работу), тогда его можно было бы снабдить такой аннотацией следующим образом:

```
@ConstructorProperties({"name", "salary", "hireDay"})
public Employee(String n, double s, Date d)
```

Данная аннотация указывала бы кодировщику на необходимость вызова методов `getName`, `getSalary` и `getHireDay` и записи результирующих значений в выражение `object`.

Аннотация `@ConstructorProperties` впервые появилась в версии Java SE 6, и пока что применялась только для классов из API-интерфейса JMX (Java Management Extensions — управляющие расширения Java).

Создание объектов с помощью фабричных методов

В некоторых случаях приходится сохранять объекты, для создания которых использовались не конструкторы, а фабричные методы. Рассмотрим, например, процесс формирования объекта `InetAddress`:

```
byte[] bytes = new byte[] { 127, 0, 0, 1};
InetAddress address = InetAddress.getByAddress(bytes);
```

В этом случае метод `instantiate()` объекта `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() });
}
```

Ниже приведен пример выходных данных.

```
<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>
```



При инсталляции делегата следует использовать конкретный подкласс, например, `Inet4Address` вместо абстрактного класса `InetAddress`.

Отслеживание действий после вызова конструктора

Состояние некоторых классов формируется путем вызова методов, не являющихся `set`-методами свойств. Отследить эту ситуацию можно, переопределив метод `initialize()` 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 этого типа нужно установить те же биты, что и в исходном объекте. Приведенный ниже метод `initialize()` генерирует необходимые 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 } ));
}
```

Ниже представлены выходные 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>
```



Может показаться более целесообразным использовать код `new Statement(bs, "set", new Object[] { i } )`, однако в этом случае `XMLWriter` генерирует сложный для восприятия оператор, устанавливающий свойство с пустым именем.

Временные свойства

Иногда в классах встречаются свойства, `get`- и `set`-методы которых распознаются `XMLDecoder`, но значения которых вы не хотите включать в архив. Для того чтобы запретить запись свойства, необходимо пометить его в составе дескриптора как *временное*, или *переходное* (`transient`). Например, приведенный ниже оператор помечает подобным образом свойство `removeMode` класса `DamageReporter` (код которого будет более детально приводиться в следующем разделе):

```
BeanInfo info = Introspector.getBeanInfo(DamageReport.class);
for (PropertyDescriptor desc : info.getPropertyDescriptors())
  if (desc.getName().equals("removeMode"))
    desc.setValue("transient", Boolean.TRUE);
```

Программа, код которой показан в листинге 8.9, демонстрирует работу различных делегатов. Учтите, что данная программа предполагает наихудшее развитие событий; в реальных приложениях многие классы могут быть записаны в архив без применения делегатов.

Листинг 8.9. Содержимое файла `PersistenceDelegateTest.java`

```
1.  import java.awt.geom.*;
2.  import java.beans.*;
3.  import java.net.*;
4.  import java.util.*;
5.  /**
6.   * Данная программа демонстрирует использование различных
7.   * делегатов по обеспечению постоянства объектов.
```

```
8.    * @version 1.01 2007-10-03
9.    * @author Cay Horstmann
10.   */
11.  public class PersistenceDelegateTest
12.  {
13.    public static class Point
14.    {
15.      @ConstructorProperties( { "x", "y" })
16.      public Point(int x, int y)
17.     {
18.      this.x = x;
19.      this.y = y;
20.     }
21.
22.     public int getX()
23.     {
24.      return x;
25.     }
26.
27.     public int getY()
28.     {
29.      return y;
30.      }
31.
32.     private final int x, y;
33.   }
34.
35.   public static void main(String[] args) throws Exception
36.   {
37.     PersistenceDelegate delegate = new PersistenceDelegate()
38.       {
39.        protected Expression instantiate(Object oldInstance, Encoder out)
40.        {
41.         Employee e = (Employee) oldInstance;
42.         GregorianCalendar c = new GregorianCalendar();
43.         c.setTime(e.getHireDay());
44.         return new Expression(oldInstance, Employee.class, "new", new Object[] {
45.           e.getName(), e.getSalary(), c.get(Calendar.YEAR),
46.           c.get(Calendar.MONTH), c.get(Calendar.DATE) });
47.        }
48.       };
49.     BeanInfo info = Introspector.getBeanInfo(Employee.class);
50.     info.getBeanDescriptor().setValue("persistenceDelegate", delegate);
51.
52.     XMLEncoder out = new XMLEncoder(System.out);
53.     out.setExceptionListener(new ExceptionListener()
54.     {
55.       public void exceptionThrown(Exception e)
56.       {
57.         e.printStackTrace();
58.       }
59.     });
60.
61.     out.setPersistenceDelegate(Rectangle2D.Double.class,
62.           new DefaultPersistenceDelegate(
63.           new String[] { "x", "y", "width", "height" }));
64.
```

```
65.     out.setPersistenceDelegate(Inet4Address.class, new DefaultPersistenceDelegate()
66.     {
67.       protected Expression instantiate(Object oldInstance, Encoder out)
68.       {
69.        return new Expression(oldInstance, InetAddress.class, "getByAddress",
70.        new Object[] { ((InetAddress) oldInstance).getAddress() });
71.       }
72.     });
73.     out.setPersistenceDelegate(BitSet.class, new DefaultPersistenceDelegate()
74.     {
75.       protected void initialize(Class<?> type, Object oldInstance,
76.          Object newInstance, Encoder out)
77.       {
78.         super.initialize(type, oldInstance, newInstance, out);
79.         BitSet bs = (BitSet) oldInstance;
80.         for (int i = bs.nextSetBit(0); i >= 0; i = bs.nextSetBit(i + 1))
81.           out.writeStatement(new Statement(bs, "set",
82.                              new Object[] { i, i + 1, true }));
83.       }
84.     });
85.     out.writeObject(new Employee("Harry Hacker", 50000, 1989, 10, 1));
86.     out.writeObject(new Point(17, 29));
87.     out.writeObject(new java.awt.geom.Rectangle2D.Double(5, 10, 20, 30));
88.     out.writeObject(InetAddress.getLocalHost());
89.     BitSet bs = new BitSet();
90.     bs.set(1, 4);
91.     bs.clear(2, 3);
92.     out.writeObject(bs);
93.     out.close();
94.   }
95. }
```

Целый пример постоянного компонента JavaBeans

Обсуждение средств обеспечения постоянства JavaBeans завершим рассмотрением конкретного примера (рис. 8.15).

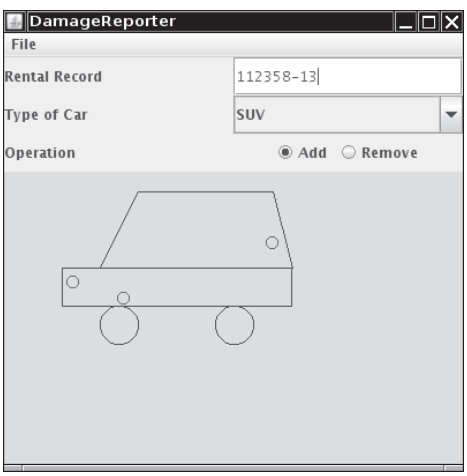


Рис. 8.15. Окно приложения `DamageReporter`

Создаваемое нами приложение записывает отчет о повреждениях арендованного автомобиля. Агент по аренде автомобилей вводит запись об аренде, выбирает тип машины, отмечает с помощью мыши поврежденные участки и сохраняет отчет. Приложение также позволяет загрузить существующий отчет. Код программы представлен в листинге 8.10.

Для сохранения и загрузки объектов `DamageReport` приложение использует средства обеспечения постоянства JavaBeans (листинг 8.11). Оно иллюстрирует ряд особенностей данной технологии.

- Свойства автоматически сохраняются и восстанавливаются. Для обеспечения хранения свойств `rentalRecord` и `carType` не требуется предпринимать никаких специальных действий.
- Для восстановления информации о поврежденных участках необходимо выполнять действия после вызова конструктора. Делегат генерирует операторы, в которых предусмотрен вызов метода `click()`.
- Для класса `Point2D.Double` нужен делегат `DefaultPersistenceDelegate`, который создает точку, используя значения свойств `x` и `y`.
- Свойство `removeMode` (оно определяет, удаляет или добавляет щелчок мыши отметку о повреждении) является временным, так как не сохраняется в отчете.

Ниже приведен пример отчета о повреждениях:

```
<?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>
```



Приложение, рассматриваемое в качестве примера, не использует средства обеспечения постоянства JavaBeans для сохранения пользовательского интерфейса. Этот вопрос может представлять интерес для создателей инструментальных средств разработки. Здесь же мы в основном уделяем внимание применению механизма постоянства для хранения данных приложения.

Этим примером мы завершаем обсуждение постоянных компонентов JavaBeans. Ниже кратко сформулированы характеристики средств обеспечения постоянства.

- Они допускают долговременное хранение.
- Они обеспечивают быстрое создание архивов, причем размеры архивов достаточно малы.
- Они просты в использовании.
- Они позволяют редактировать архивы вручную.
- Они входят в состав стандартного набора средств Java.

Листинг 8.10. Содержимое файла `DamageReporter.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.awt.geom.*;
4.  import java.beans.*;
5.  import java.io.*;
6.  import java.util.*;
7.  import javax.swing.*;
8.
9.  /**
10.  * Данная программа демонстрирует использование объектов
11.  * XMLEncoder и XMLDecoder. В составе класса объединены
12.  * пользовательский интерфейс и средства рисования.
13.  * Заслуживают внимания обработчики событий для openItem и
14.  * saveItem. В классе DamageReport приведен пример настройки
15.  * средств кодирования.
16.  * @version 1.01 2004-10-03
17.  * @author Cay Horstmann
18.  */
19. public class DamageReporter extends JFrame
20. {
21.    public static void main(String[] args)
22.    {
23.       JFrame frame = new DamageReporter();
24.       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25.       frame.setVisible(true);
26.    }
27.
28.    public DamageReporter()
29.    {
30.       setTitle("DamageReporter");
31.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
32.
33.       chooser = new JFileChooser();
34.       chooser.setCurrentDirectory(new File("."));
35.
36.       report = new DamageReport();
37.       report.setCarType(DamageReport.CarType.SEDAN);
38.
39.       // Создание строки меню.
40.       JMenuBar menuBar = new JMenuBar();
41.       setJMenuBar(menuBar);
42.
43.       JMenu menu = new JMenu("File");
```

```
44.       menuBar.add(menu);
45.
46.       JMenuItem openItem = new JMenuItem("Open");
47.       menu.add(openItem);
48.       openItem.addActionListener(new
49.         ActionListener()
50.         {
51.           public void actionPerformed(ActionEvent evt)
52.           {
53.             // Отображение диалогового окна выбора файла.
54.             int r = chooser.showOpenDialog(null);
55.
56.             // Открытие выбранного файла.
57.             if(r == JFileChooser.APPROVE_OPTION)
58.             {
59.               try
60.               {
61.                 File file = chooser.getSelectedFile();
62.                 XMLDecoder decoder = new
63.                    XMLDecoder(new FileInputStream(file));
64.                 report = (DamageReport) decoder.readObject();
65.                 decoder.close();
66.                 rentalRecord.setText(report.getRentalRecord());
67.                 carType.setSelectedItem(report.getCarType());
68.                 repaint();
69.               }
70.               catch (IOException e)
71.               {
72.                 JOptionPane.showMessageDialog(null, e);
73.               }
74.             }
75.           }
76.         });
77.
78.       JMenuItem saveItem = new JMenuItem("Save");
79.       menu.add(saveItem);
80.       saveItem.addActionListener(new
81.         ActionListener()
82.         {
83.           public void actionPerformed(ActionEvent evt)
84.           {
85.             report.setRentalRecord(rentalRecord.getText());
86.             chooser.setSelectedFile(new
87.               File(rentalRecord.getText() + ".xml"));
88.
89.             // Отображение диалогового окна выбора файла.
90.             int r = chooser.showSaveDialog(null);
91.
92.             // Если файл выбран, выполняется сохранение.
93.             if(r == JFileChooser.APPROVE_OPTION)
94.             {
95.               try
96.               {
97.                 File file = chooser.getSelectedFile();
98.                 XMLEncoder encoder = new
99.                   XMLEncoder(new FileOutputStream(file));
100.                report.configureEncoder(encoder);
```

```
101.                encoder.writeObject(report);
102.                encoder.close();
103.             }
104.             catch (IOException e)
105.             {
106.                JOptionPane.showMessageDialog(null, e);
107.             }
108.          }
109.       }
110.    });
111.
112.    JMenuItem exitItem = new JMenuItem("Exit");
113.    menu.add(exitItem);
114.    exitItem.addActionListener(new
115.       ActionListener()
116.       {
117.          public void actionPerformed(ActionEvent event)
118.          {
119.             System.exit(0);
120.          }
121.       });
122.
123.    // Раскрывающийся список для выбора типа автомобиля.
124.    rentalRecord = new JTextField();
125.    carType = new JComboBox();
126.    carType.addItem(DamageReport.CarType.SEDAN);
127.    carType.addItem(DamageReport.CarType.WAGON);
128.    carType.addItem(DamageReport.CarType.SUV);
129.
130.    carType.addActionListener(new ActionListener()
131.       {
132.          public void actionPerformed(ActionEvent event)
133.          {
134.             DamageReport.CarType item =
135.                (DamageReport.CarType)carType.getSelectedItem();
136.             report.setCarType(item);
137.             repaint();
138.          }
139.       });
140.
141.    // Компонент для отображения состояния автомобиля
142.    // и мест повреждений.
143.    carComponent = new JComponent()
144. {
145.   public void paintComponent(Graphics g)
146.   {
147.    Graphics2D g2 = (Graphics2D) g;
148.    g2.setColor(new Color(0.9f, 0.9f, 0.45f));
149.    g2.fillRect(0, 0, getWidth(), getHeight());
150.    g2.setColor(Color.BLACK);
151.   g2.draw(shapes.get(report.getCarType()));
152.   report.drawDamage(g2);
153.   }
154. };
155. carComponent.addMouseListener(new MouseAdapter()
156. {
```

```
157.      public void mousePressed(MouseEvent event)
158.      {
159.       report.click(new Point2D.Double(event.getX(), event.getY()));
160.       repaint();
161.      }
162.    });
163.
164.      // Переключатели для выбора действий по щелчку кнопкой мыши.
165.      addButton = new JRadioButton("Add");
166.      removeButton = new JRadioButton("Remove");
167.      ButtonGroup group = new ButtonGroup();
168.      JPanel buttonPanel = new JPanel();
169.      group.add(addButton);
170.      buttonPanel.add(addButton);
171.      group.add(removeButton);
172.      buttonPanel.add(removeButton);
173.      addButton.setSelected(!report.getRemoveMode());
174.      removeButton.setSelected(report.getRemoveMode());
175.      addButton.addActionListener(new
176.        ActionListener()
177.        {
178.          public void actionPerformed(ActionEvent event)
179.          {
180.            report.setRemoveMode(false);
181.          }
182.        });
183.      removeButton.addActionListener(new
184.        ActionListener()
185.        {
186.          public void actionPerformed(ActionEvent event)
187.          {
188.            report.setRemoveMode(true);
189.          }
190.        });
191.
192.      // Размещение компонентов.
193.      JPanel gridPanel = new JPanel();
194.      gridPanel.setLayout(new GridLayout(0, 2));
195.      gridPanel.add(new JLabel("Rental Record"));
196.      gridPanel.add(rentalRecord);
197.      gridPanel.add(new JLabel("Type of Car"));
198.      gridPanel.add(carType);
199.      gridPanel.add(new JLabel("Operation"));
200.      gridPanel.add(buttonPanel);
201.
202.      add(gridPanel, BorderLayout.NORTH);
203.      add(carPanel, BorderLayout.CENTER);
204.    }
205.
206.    private JTextField rentalRecord;
207.    private JComboBox carType;
208.    private JComponent carComponent;;
209.    private JRadioButton addButton;
210.    private JRadioButton removeButton;
211.    private DamageReport report;
212.    private JFileChooser chooser;
213.
```

```
214.   private static final int DEFAULT_WIDTH = 400;
215.   private static final int DEFAULT_HEIGHT = 400;
216.
217.   private static Map<DamageReport.CarType, Shape> shapes =
         new EnumMap<DamageReport.CarType, Shape>
218.      (DamageReport.CarType.class);
219.
220.   static
221.   {
222.      int width = 200;
223.      int x = 50;
224.      int y = 50;
225.      Rectangle2D.Double body = new
226.          Rectangle2D.Double(x, y + width / 6, width - 1, width / 6);
227.      Ellipse2D.Double frontTire = new
228.        Ellipse2D.Double(x + width / 6, y + width / 3,
229.        width / 6, width / 6);
230.      Ellipse2D.Double rearTire = new
231.        Ellipse2D.Double(x + width * 2 / 3, y + width / 3,
232.        width / 6, width / 6);
233.
234.      Point2D.Double p1 = new
235.        Point2D.Double(x + width / 6, y + width / 6);
236.      Point2D.Double p2 = new Point2D.Double(x + width / 3, y);
237.      Point2D.Double p3 = new
238.        Point2D.Double(x + width * 2 / 3, y);
239.      Point2D.Double p4 = new
240.        Point2D.Double(x + width * 5 / 6, y + width / 6);
241.
242.      Line2D.Double frontWindshield = new Line2D.Double(p1, p2);
243.      Line2D.Double roofTop = new Line2D.Double(p2, p3);
244.      Line2D.Double rearWindshield = new Line2D.Double(p3, p4);
245.
246.      GeneralPath sedanPath = new GeneralPath();
247.      sedanPath.append(frontTire, false);
248.      sedanPath.append(rearTire, false);
249.      sedanPath.append(body, false);
250.      sedanPath.append(frontWindshield, false);
251.      sedanPath.append(roofTop, false);
252.      sedanPath.append(rearWindshield, false);
253.      shapes.put(DamageReport.CarType.SEDAN, sedanPath);
254.
255.      Point2D.Double p5 = new
256.        Point2D.Double(x + width * 11 / 12, y);
257.      Point2D.Double p6 = new
258.        Point2D.Double(x + width, y + width / 6);
259.      roofTop = new Line2D.Double(p2, p5);
260.      rearWindshield = new Line2D.Double(p5, p6);
261.
262.      GeneralPath wagonPath = new GeneralPath();
263.      wagonPath.append(frontTire, false);
264.      wagonPath.append(rearTire, false);
265.      wagonPath.append(body, false);
266.      wagonPath.append(frontWindshield, false);
267.      wagonPath.append(roofTop, false);
268.      wagonPath.append(rearWindshield, false);
269.      shapes.put(DamageReport.CarType.WAGON, wagonPath);
```

```
270.      Point2D.Double p7 = new
271.        Point2D.Double(x + width / 3, y - width / 6);
272.      Point2D.Double p8 = new
273.        Point2D.Double(x + width * 11 / 12, y - width / 6);
274.      frontWindshield = new Line2D.Double(p1, p7);
275.      roofTop = new Line2D.Double(p7, p8);
276.      rearWindshield = new Line2D.Double(p8, p6);
277.
278.      GeneralPath suvPath = new GeneralPath();
279.      suvPath.append(frontTire, false);
280.      suvPath.append(rearTire, false);
281.      suvPath.append(body, false);
282.      suvPath.append(frontWindshield, false);
283.      suvPath.append(roofTop, false);
284.      suvPath.append(rearWindshield, false);
285.      shapes.put(DamageReport.CarType.SUV, suvPath);
286.   }
287. }
```

Листинг 8.11. Содержимое файла `DamageReport.java`

```
1.  import java.awt.*;
2.  import java.awt.geom.*;
3.  import java.beans.*;
4.  import java.util.*;
5.
6.  /**
7.   * Данный класс описывает отчет о повреждениях транспортного средства,
8.   * который можно сохранять и загружать впоследствии. Механизм записи
9.   * обеспечивает долговременное хранение.
10.  * @version 1.21 2004-08-30
11.  * @author Cay Horstmann
12.  */
13. public class DamageReport
14. {
15.   public enum CarType
16.   {
17.    SEDAN, WAGON, SUV
18.   }
19.
20.   // Данное свойство сохраняется автоматически.
21.   public void setRentalRecord(String newValue)
22.   {
23.     rentalRecord = newValue;
24.   }
25.
26.   public String getRentalRecord()
27.   {
28.     return rentalRecord;
29.   }
30.
31.   // Данное свойство сохраняется автоматически.
32.   public void setCarType(CarType newValue)
33.   {
34.     carType = newValue;
35.   }
36.
```

```
37.    public CarType getCarType()
38.    {
39.       return carType;
40.    }
41.
42.    // Данное свойство объявлено как временное.
43.    public void setRemoveMode(boolean newValue)
44.    {
45.       removeMode = newValue;
46.    }
47.
48.    public boolean getRemoveMode()
49.    {
50.       return removeMode;
51.    }
52.
53.    public void click(Point2D p)
54.    {
55.       if (removeMode)
56.       {
57.          for (Point2D center : points)
58.          {
59.             Ellipse2D circle = new Ellipse2D.Double(
60.                center.getX() - MARK_SIZE,
61.                center.getY() - MARK_SIZE,
62.                2 * MARK_SIZE, 2 * MARK_SIZE);
63.             if (circle.contains(p))
64.             {
65.                points.remove(center);
66.                return;
67.             }
68.          }
69.       }
70.       else points.add(p);
71.    }
72.
73.    public void drawDamage(Graphics2D g2)
74.    {
75.       g2.setPaint(Color.RED);
76.       for (Point2D center : points)
77.       {
78.          Ellipse2D circle = new Ellipse2D.Double(
79.             center.getX() - MARK_SIZE, center.getY() - MARK_SIZE,
80.             2 * MARK_SIZE, 2 * MARK_SIZE);
81.          g2.draw(circle);
82.       }
83.    }
84.
85.    public void configureEncoder(XMLEncoder encoder)
86.    {
87.       // Данный фрагмент необходим для сохранения
88.       // объектов Point2D.Double.
89.       encoder.setPersistenceDelegate(
90.          Point2D.Double.class, new
91.          DefaultPersistenceDelegate(new String[]{ "x", "y" }) );
92.
```

```
93.       // Данный фрагмент необходим, потому что списочный
94.       // массив точек не представляется как свойство.
95.       encoder.setPersistenceDelegate(DamageReport.class, new
96.            DefaultPersistenceDelegate()
97.            {
98.              protected void initialize(Class<?> type,
99.                 Object oldInstance, Object newInstance,
100.                Encoder out)
101.             {
102.                super.initialize(type, oldInstance,
103.                  newInstance, out);
104.               DamageReport r = (DamageReport) oldInstance;
105.
106.               for (Point2D p : r. points)
107.                  out.writeStatement(new
108.                       Statement(oldInstance,"click",
109.                         new Object[]{ p }) );
110.           }
111.         });
112.
113.   }
114.   // Данный фрагмент необходим, чтобы объявить
115.   // свойство removeMode как временное.
116.   static
117.   {
118.     try
119.     {
120.       BeanInfo info =
121.         Introspector.getBeanInfo(DamageReport.class);
122.       for (PropertyDescriptor desc :
123.         info.getPropertyDescriptors())
124.         if (desc.getName().equals("removeMode"))
125.           desc.setValue("transient", Boolean.TRUE);
126.     }
127.     catch (IntrospectionException e)
128.     {
129.       e.printStackTrace();
130.     }
131.   }
132.
133.   private String rentalRecord;
134.   private CarType carType;
135.   private boolean removeMode;
136.   private ArrayList<Point2D> points = new ArrayList<Point2D>();
137.
138.   private static final int MARK_SIZE = 5;
139. }
```

API java.beans.XMLEncoder 1.4

- `XMLEncoder(OutputStream out)`
  Создает объект `XMLEncoder`, который записывает выходные данные в указанный поток.
- `void writeObject(Object obj)`
  Записывает в архив указанный объект.

- `void writeStatement(Statement stat)`

  Записывает в архив указанный оператор. Этот метод вызывается только делегатом.

API java.beans.Encoder 1.4

- `void setPersistenceDelegate(Class<?> type, PersistenceDelegate delegate)`
- `PersistenceDelegate getPersistenceDelegate(Class<?> type)`

  Устанавливает или возвращает архивированный объект указанного типа.

- `void setExceptionListener(ExceptionListener listener)`
- `ExceptionListener getExceptionListener()`

  Устанавливает или возвращает обработчик, который уведомляется о том, что в процессе кодирования возникло исключение.

API java.beans.ExceptionListener 1.4

- `void exceptionThrown(Exception e)`

  Данный метод вызывается, если в процессе кодирования или декодирования возникло исключение.

API java.beans.XMLDecoder 1.4

- `XMLDecoder(InputStream in)`

  Создает объект `XMLDecoder`, который осуществляет чтение архива из указанного входного потока.

- `Object readObject()`

  Читает очередной объект из архива.

- `void setExceptionListener(ExceptionListener listener)`
- `ExceptionListener getExceptionListener()`

  Устанавливает или возвращает обработчик, который оповещается о возникновении исключения в процессе кодирования.

API java.beans.PersistenceDelegate 1.4

- `protected abstract Expression instantiate(Object oldInstance, Encoder out)`

  Возвращает выражение для создания экземпляра объекта, эквивалентного `oldInstance`.

- `protected void initialize(Class<?> type, Object oldInstance, Object newInstance, Encoder out)`

  Записывает в `out` операторы, которые возвращают в `newInstance` ссылку на объект, эквивалентный `oldInstance`.

API java.beans.DefaultPersistenceDelegate 1.4

- `DefaultPersistenceDelegate()`

  Создает делегат для класса с конструктором без параметров.

- `DefaultPersistenceDelegate(String[] propertyNames)`

  Создает делегат для класса, параметры конструктора которого являются значениями указанных свойств.

- `protected Expression instantiate(Object oldInstance, Encoder out)`

  Возвращает выражение для вызова конструктора либо без параметров, либо с указанием значений свойств.

- `protected void initialize(Class<?> type, Object oldInstance, Object newInstance, Encoder out)`

  Записывает в `out` операторы, которые применяют `set`-методы свойств к `newInstance`, пытаясь сделать объект эквивалентным `oldInstance`.

API java.beans.Expression 1.4

- `Expression(Object value, Object target, String methodName, Object[] parameters)`

  Создает выражение, которое вызывает указанный метод `target` с указанными параметрами. Предполагается, что результатом выполнения выражения является `value`. Для того чтобы конструктор мог быть вызван, параметр `target` должен представлять собой объект `Class`, а строка `methodName` должна содержать `new`.

API java.beans.Statement 1.4

- `Statement(Object target, String methodName, Object[] parameters)`

  Создает оператор, который вызывает указанный метод `target` с указанными параметрами.

На этом три длинных главы по программированию GUI с помощью Swing, AWT и JavaBeans завершены. В следующей главе речь пойдет уже о совершенно иной теме, а именно — о безопасности. Безопасность всегда была ключевой характеристикой платформы Java. С постоянным увеличением степени риска в мире, где мы живем и работаем, детальные знания в области средств безопасности Java будут полезны многим разработчикам.

Глава 9

СРЕДСТВА ЗАЩИТЫ

В ЭТОЙ ГЛАВЕ...

- Загрузчики классов
- Верификация байтового 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, доставленных через Internet (более подробно аплеты рассматривались в главе 10 первого тома). Понятно, что доставка исполняемых аплетов имеет смысл только тогда, когда получатели уверены, что код не может нанести ущерб 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 позволяют обеспечивать три следующих механизма.

- Структурные функциональные возможности языка (вроде проверки границ массивов, не выполнения преобразования непроверенных типов, отсутствия указателей и т.д.).
- Средства контроля доступа, следящие за тем, какие действия коду можно выполнять, а какие нельзя (например, может ли код получать доступ к файлам, передавать данные по сети и т.д.).
- Механизм цифровой подписи, предоставляющий авторам кодов возможность применять стандартные алгоритмы для аутентификации своих продуктов, а пользователям — затем точно определять, кто создал код и изменился ли он с момента его подписания.

В этой главе сначала речь пойдет о *загрузчиках классов* (class loader), проверяющих файлы классов на предмет целостности при их загрузке в виртуальную машину, и том, как этот механизм может выявлять в файлах классов признаки фальсификации.

Для обеспечения максимальной безопасности как используемый по умолчанию механизм для загрузки классов, так и специальный загрузчик классов должны оба взаимодействовать с классом *диспетчера защиты* (security manager), следящим за тем, какие действия коду разрешено выполнять, а какие — нет. Поэтому далее в этой главе будет довольно подробно показано, как можно настраивать безопасность для платформы Java.

И, наконец, в завершение главы будут рассмотрены различные поставляемые в пакете `java.security` алгоритмы шифрования, позволяющие подписывать код и обеспечивать аутентификацию пользователей.

Как обычно, основное внимание здесь будет уделяться только тем вопросам, которые представляют наибольший интерес для разработчиков приложений. Тем, кому требуется получить более глубокое представление по данной теме, рекомендуем прочитать книгу Ли Гонга (Li Gong), Гэри Эллисона (Gary Ellison) и Мэри Дейджфорд (Mary Dageforde) *Inside Java 2 Platform Security: Architecture, API Design, and Implementation, 2nd ed* (Prentice Hall PTR, 2003 г.).

Загрузчики классов

Компилятор Java преобразует исходные инструкции в понятный для виртуальной машины Java код. Этот код сохраняется в файле класса с расширением `.class`. В каждом таком файле класса содержится код определения и реализации для только одного класса или интерфейса. Далее все эти файлы классов интерпретируются программой, которая умеет преобразовывать набор инструкций виртуальной машины в машинные команды используемого компьютера.

Учтите, что интерпретатор виртуальной машины загружает только те файлы классов, которые необходимы для выполнения программы в данный момент. Допустим, что 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 считывания их с диска или копирования по сети. С помощью этого механизма загружается содержимое файла класса `MyProgram`.
2. Если в классе `MyProgram` встречаются поля или объекты, ссылающиеся на классы других типов, тогда также еще загружаются и файлы этих классов. (Процесс загрузки всех классов, от которых зависит данный класс, называется *разрешением* (*resolving*) класса.)
3. Затем виртуальная машина выполняет метод `main()` класса `MyProgram`. (Этот метод является статическим, поэтому никаких экземпляров класса `MyProgram` создавать не требуется.)
4. Если для работы метода `main()` или метода, вызываемого из `main()`, требуются дополнительные классы, то загружаются соответствующие им файлы.

Следует отметить, что механизм загрузки классов использует не один, а несколько загрузчиков. С каждой программой Java применяется, по крайней мере, три загрузчика классов:

- *первичный загрузчик классов* (bootstrap class loader);
- *расширенный загрузчик классов* (extension class loader);
- *системный загрузчик классов* (system class loader), иногда называемый *прикладным загрузчиком классов* (application class loader).

Первичный загрузчик классов загружает системные классы, как правило, из файла `rt.jar`. Он является неотъемлемой частью виртуальной машины и обычно реализуется на языке C. Первичному загрузчику классов не соответствует ни одного объекта `ClassLoader`. Например, показанный ниже вызов возвращает значение `null`:

```
String.class.getClassLoader()
```

Расширенный загрузчик классов загружает стандартное расширение из каталога *jre*`/lib/ext`. Если переместить файлы `.jar` в этот каталог, то расширенный загрузчик классов найдет в нем нужные классы расширения даже без указания пути к классам. (Некоторые специалисты рекомендуют применять этот механизм, чтобы избежать путаницы с указанием пути к классам, но такой способ связан с определенными сложностями, которые описаны в приведенной ниже врезке “Внимание”.)

Системный загрузчик классов загружает прикладные классы, которые размещаются в каталогах и файлах формата JAR/ZIP. Путь к классам должен быть указан с помощью переменной окружения `CLASSPATH` или параметра командной строки — `classpath`.

В системе, реализованной компанией Sun, расширенный и системный загрузчики реализуются на языке Java. Оба они представляют собой экземпляры класса `URLClassLoader`.



Помещать библиотеки в каталог *jre*`/lib/ext` нежелательно по следующей причине. Предположим, что вам понадобился один из классов этой библиотеки, который, в свою очередь, должен загружать класс, не являющийся системным или расширенным. Расширенный загрузчик классов *не использует информацию о пути к классам, устанавливаемую в переменной окружения* `CLASSPATH`. Этот факт необходимо учитывать, принимая решения об использовании каталога *jre*`/lib/ext` в качестве хранилища для своих классов.



Помимо всех уже перечисленных мест, классы еще также могут загружаться и из каталога *jre*/lib/endorsed. Такой механизм может применяться только для замены определенных стандартных библиотек Java (вроде тех, что предназначены для обеспечения поддержки XML и CORBA) более новыми версиями. Подробнее об этом можно узнать по адресу http://java.sun.com/javase/6/docs/technotes/guides/standards/index.html.

Иерархия загрузчиков классов

Загрузчики классов связаны отношениями *родитель/потомок*. У каждого загрузчика классов, за исключением первичного, имеется родительский загрузчик классов. Предполагается, что загрузчик классов дает шанс своему родителю загружать любой нужный класс и загружает его сам только в том случае, если этого не может сделать родитель. Например, если системный загрузчик классов запрашивает загрузку системного класса (например, java.util.arrayList), то прежде он предлагает загрузить его расширенному загрузчику классов. Расширенный загрузчик классов, в свою очередь, предлагает сделать это первичному загрузчику классов. Наконец, первичный загрузчик классов находит и загружает класс из файла rt.jar; при этом все другие загрузчики классов прекращают дальнейший поиск.

Некоторые программы имеют основанную на подключаемых модулях архитектуру, в которой определенные части кода упаковываются в виде дополнительных необязательных модулей. Если подключаемые модули упаковываются в виде файлов JAR, классы этих модулей могут загружаться просто с помощью экземпляра URLClassLoader:

```
URL url = new URL("file:///path/to/plugin.jar");
URLClassLoader pluginLoader = new URLClassLoader(new URL[] { url });
Class<?> cl = pluginLoader.loadClass("mypackage.MyClass");
```

Здесь, поскольку никакой родитель не был указан в конструкторе URLClassLoader, в роли родителя pluginLoader будет выступать системный загрузчик классов. Общий вид иерархии загрузчиков классов показан на рис. 9.1.

В большинстве случае разработчику не приходится волноваться об их иерархии загрузчиков классов. Как правило, классы загружаются потому, что этого требуют другие классы, и этот процесс является абсолютно прозрачным для разработчика.

Однако иногда разработчику все-таки бывает нужно вмешиваться и указывать загрузчик класса. Рассмотрим следующий пример.

- В коде приложения содержится вспомогательный метод, вызывающий Class.forName(classNameString).
- Этот метод вызывается из класса подключаемого модуля.
- Параметр classNameString указывает на класс, содержащийся в plugin.jar.

Автор подключаемого модуля имеет все основания ожидать, что этот класс будет загружаться. Однако класс вспомогательного метода будет загружать системный загрузчик классов, а это именно тот загрузчик, который необходим Class.forNam. Это значит, что классы в plugin.jar видны не будут. Данный феномен называется *инверсией загрузчиков классов* (classloader inversion).

Для устранения этой проблемы необходимо, чтобы вспомогательный метод использовал правильный загрузчик классов. Это может потребовать либо указания загрузчика классов в виде параметра, либо установки нужного загрузчика в качестве контекстного загрузчика классов (context class loader) текущего потока. Вторая стратегия применяется очень во многих инфраструктурах (например, в инфраструктуре JAXP и инфраструктуре JNDI, о которых рассказывалось в главах 2 и 4).

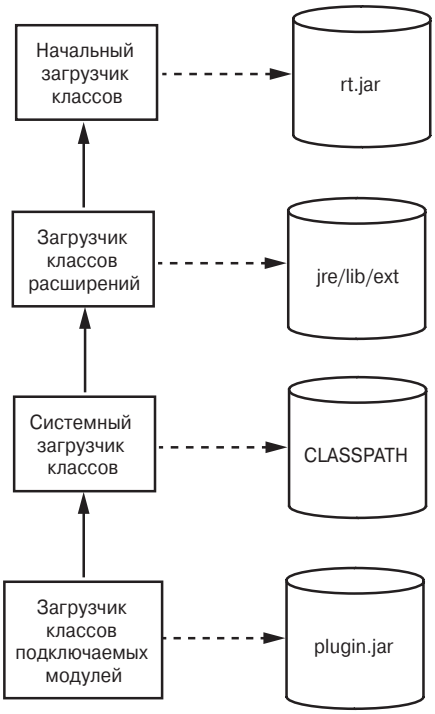


Рис. 9.1. Иерархия загрузчиков классов

У каждого потока имеется ссылка на загрузчик классов, называемый контекстным загрузчиком классов. Контекстным загрузчиком главного потока является системный загрузчик классов. При создании нового потока ему в качестве контекстного загрузчика назначается контекстный загрузчик классов того потока, который его создает. Следовательно, если ничего не делать, всем потокам в качестве контекстного загрузчика будет назначаться системный загрузчик классов.

Однако также существует и возможность указывать любой другой загрузчик классов путем вызова:

```
Thread t = Thread.currentThread();
t.setContextClassLoader(loader);
```

В рассматриваемом примере это означает, что вспомогательный метод тогда может извлекать контекстный загрузчик классов:

```
Thread t = Thread.currentThread();
ClassLoader loader = t.getContextClassLoader();
Class cl = loader.loadClass(className);
```

Теперь остается только вопрос о том, когда этому контекстному загрузчику классов должен назначаться загрузчик класса подключаемого модуля. Данное решение зависит уже от самого разработчика приложения. Обычно неплохим вариантом считается делать так, чтобы контекстный загрузчик классов устанавливался при вызове метода класса подключаемого модуля, который был загружен с помощью другого загрузчика классов. В качестве альтернативного варианта, однако, можно делать и так, чтобы контекстный загрузчик классов устанавливала вызывающая программа вспомогательного метода.



В случае написания метода, загружающего класс по имени, рекомендуется предоставлять вызывающей программе возможность выбирать между передачей явного загрузчика классов и применением контекстного загрузчика классов. Просто использовать загрузчик класса метода не стоит.

Использование загрузчиков классов в качестве пространств имен

Каждый Java-программист знает, что для исключения вероятности возникновения конфликтов между именами применяются имена пакетов. Например, в стандартной библиотеке Java есть два класса с именем `Date`, но полностью их имена, конечно же, выглядят как `java.util.Date` и `java.sql.Date`. Короткие имена существуют для удобства программиста и требуют включения соответствующих операторов `import`. В выполняющейся программе же все имена классов обязательно содержат имена своих пакетов.

Как ни странно, но в рамках 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 Web-браузере для каждой Web-страницы используются отдельные экземпляры загрузчика классов аплетов. Это позволяет виртуальной машине различать классы разных Web-страниц независимо от их имен. На рис. 9.2 показан пример. Предположим, Web-страница содержит два аплета, предоставляемых двумя разными рекламодателями, и у каждого из этих аплетов имеется класс с именем `Banner`. Поскольку каждый аплет загружается отдельным загрузчиком классов, эти классы являются совершенно отдельными и не конфликтуют друг с другом.



Существуют и другие способы использования данной технологии, например, "горячее развертывание" сервлетов и компонентов Enterprise JavaBeans. Более подробную информацию по этой теме можно найти по адресу `http://java.sun.com/developer/TechTips/2000/tt1027.html`.

Создание собственного загрузчика классов

Для специальных целей можно создавать и свои собственные загрузчики классов. Такой подход позволяет выполнять перед передачей байтового кода виртуальной машине какие-то специальные проверки. Например, можно создать загрузчик классов, способный запрещать загрузку класса, который не был помечен как "оплаченный".

Создание собственного загрузчика классов осуществляется просто путем расширения класса `ClassLoader` и переопределения метода `findClass(String className)`.

Метод `loadClass()` суперкласса `ClassLoader` заботится о делегировании функций загрузки классов родителю и вызывает метод `findClass()` только в том случае, если класс еще не был загружен, и если родительскому загрузчику классов его загрузить не удалось.

Поэтому при самостоятельной реализации этого метода необходимо делать так, чтобы он выполнял следующие действия.

1. Загружал байтовый код класса из локальной файловой системы или какого-нибудь другого источника.
2. Вызывал метод `defineClass()` суперкласса `ClassLoader` для представления байтового кода виртуальной машине.

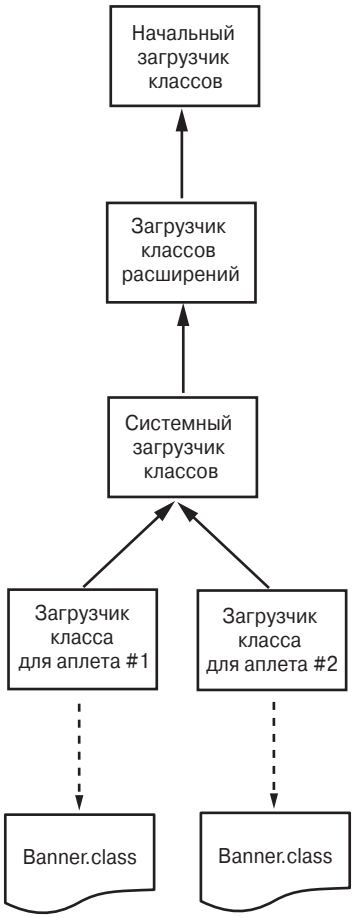


Рис. 9.2. Два загрузчика классов загружают разные классы с одинаковыми именами

В приведенном в листинге 9.1 примере программы реализуется загрузчик классов, способный загружать файлы зашифрованных классов. Эта программа запрашивает у пользователя имя первого подлежащего загрузке класса (т.е. класса, содержащего метод `main()`) и ключ для 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()` загруженного класса (как показано на рис. 9.3).

Во избежание неоправданного усложнения примера мы игнорируем более чем 2000-летний опыт в области криптографии и используем для шифрования файлов классов древний шифр Цезаря.



В прекрасной книге Дэвида Кана (David Kahn) *The Codebreakers* (Macmillan, 1967 г.) говорится, что Цезарь кодировал 24 буквы латинского алфавита, сдвигая их на 3 буквы, чем повергал в недоумение всех своих противников. Во время написания данной книги правительство США ограничивало экспорт наиболее сложных методов шифрования. Поэтому использование довольно простого метода Цезаря в нашем примере не нарушит экспортных ограничений.

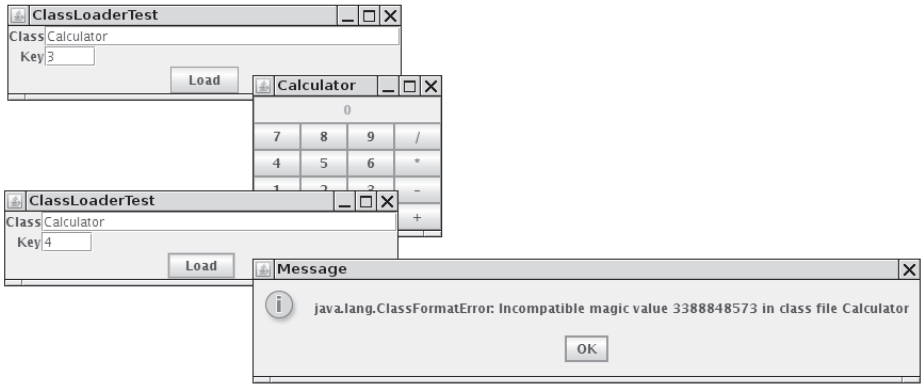


Рис. 9.3. Программа `ClassLoaderTest`

В данной версии тайнописи Цезаря в качестве ключа используется число между 1 и 255. Для шифрования байт просто суммируется с этим ключом по модулю 256. Алгоритм шифрования реализован в программе `Caesar.java`, код которой показан 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 для данной книги предлагается четыре файла классов, зашифрованных с помощью традиционного значения ключа 3. Эти классы нельзя загрузить посредством стандартного интерпретатора байт-кода, но можно запустить, применяя пользовательский загрузчик классов, реализованный в программе `ClassLoaderTest`.

Зашифрованные файлы классов имеют самое разное практическое применение (если, конечно, используется более сложный шифр, чем шифр Цезаря). Без ключа шифрования эти файлы классов бесполезны, поскольку не могут выполняться с помощью стандартной виртуальной машины, и восстановить их исходный текст нелегко.

Это значит, что для аутентификации пользователя класса или для проверки того, что программа оплачена до запуска, можно применять пользовательский загрузчик классов. Естественно, шифрование является только одним из возможных применений пользовательского загрузчика классов. Другие типы загрузчиков можно применять для решения иных проблем, например, для сохранения файлов классов в базе данных.

Листинг 9.1. Содержимое файла `ClassLoaderTest.java`

```
1.  import java.io.*;
2.  import java.lang.reflect.*;
3.  import java.awt.*;
4.  import java.awt.event.*;
5.  import javax.swing.*;
6.
7.  /**
8.   * Эта программа демонстрирует пример пользовательского
9.   * загрузчика, умеющего расшифровывать файлы классов.
10.  * @version 1.22 2007-10-05
11.  * @author Cay Horstmann
12.  */
13. public class ClassLoaderTest
14. {
```

```
15.    public static void main(String[] args)
16.    {
17.       EventQueue.invokeLater(new Runnable()
18.        {
19.          public void run()
20.          {
21.           JFrame frame = new ClassLoaderFrame();
22.           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.           frame.setVisible(true);
24.          }
25.       });
26.     }
27.  }
28.
29.  /**
30.    * Фрейм с двумя полями редактирования для ввода
31.    * имени загружаемого файла и ключа шифрования.
32.    */
33.
34.  class ClassLoaderFrame extends JFrame
35.  {
36.    public ClassLoaderFrame()
37.    {
38.      setTitle("ClassLoaderTest");
39.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
40.      setLayout(new GridBagLayout());
41.       add(new JLabel("Class"), new GBC(0, 0).setAnchor(GBC.EAST));
42.       add(nameField, new GBC(1, 0).setWeight(100,
43.          0).setAnchor(GBC.WEST));
44.       add(new JLabel("Key"), new GBC(0, 1).setAnchor(GBC.EAST));
45.       add(keyField, new GBC(1, 1).setWeight(100,
46.          0).setAnchor(GBC.WEST));
47.       JButton loadButton = new JButton("Load");
48.       add(loadButton, new GBC(0, 2, 2, 1));
49.       loadButton.addActionListener(new ActionListener()
50.         {
51.           public void actionPerformed(ActionEvent event)
52.           {
53.             runClass(nameField.getText(), keyField.getText());
54.           }
55.       });
56.    pack();
57.
58.     /**
59.      * Запуск метода main() указанного класса.
60.      * @param name Имя класса
61.      * @param key Ключ шифрования для файла класса
62.      */
63.     public void runClass(String name, String key)
64.     {
65.       try
66.       {
67.         ClassLoader loader =
68.            Class<?> c = loader.loadClass(name);
69.            Method m = c.getMethod("main", String[].class);
70.            m.invoke(null, (Object) new String[] {});
71.       }
```

```
72.       catch (Throwable e)
73.       {
74.          JOptionPane.showMessageDialog(this, e);
75.       }
76.    }
77.
78.    private JTextField keyField = new JTextField("3", 4);
79.    private JTextField nameField = new JTextField("Calculator", 30);
80.    private static final int DEFAULT_WIDTH = 300;
81.    private static final int DEFAULT_HEIGHT = 200;
82. }
83.
84. /**
85.  * Загрузчик для зашифрованных файлов классов.
86.  */
87. class CryptoClassLoader extends ClassLoader
88. {
89.    /**
90.     * Конструктор зашифрованных классов.
91.     * @param k Ключ шифрования
92.     */
93.    public CryptoClassLoader(int k)
94.    {
95.       key = k;
96.    }
97.
98.    protected Class<?> findClass(String name) throws ClassNotFoundException
99.    {
100.      byte[] classBytes = null;
101.      try
102.      {
103.         classBytes = loadClassBytes(name);
104.      }
105.      catch (IOException e)
106.      {
107.         throw new ClassNotFoundException(name);
108.      }
109.
110.      Class<?> cl = defineClass(name, classBytes, 0,
111.         classBytes.length);
112.      if (cl == null)throw new ClassNotFoundException(name);
113.      return cl;
114.   }
115.
116.   /**
117.    * Загрузка и декодирование файла класса.
118.    * @param name Имя файла класса
119.    * @return Массив с содержимым файла класса
120.    */
121.   private byte[] loadClassBytes(String name)
122.      throws IOException
123.   {
124.      String cname = name.replace('.', '/') + ".caesar";
125.      FileInputStream in = null;
126.      in = new FileInputStream(cname);
127.      try
128.      {
```

```
129.         ByteArrayOutputStream buffer = new ByteArrayOutputStream();
130.         int ch;
131.         while ((ch = in.read()) != -1)
132.         {
133.           byte b = (byte) (ch - key);
134.           buffer.write(b);
135.         }
136.         in.close();
137.         return buffer.toByteArray();
138.       }
139.       finally
140.       {
141.         in.close();
142.       }
143.    }
144.    private int key;
145. }
```

Листинг 9.2. Содержимое файла `Caesar.java`

```
1.    import java.io.*;
2.
3.   /**
4.    * Кодирование файла с использованием шифра Цезаря.
5.    * @version 1.00 1997-09-10
6.    * @author Cay Horstmann
7.    */
8.   public class Caesar
9.   {
10.    public static void main(String[] args)
11.    {
12.      if (args.length != 3)
13.      {
14.        System.out.println("USAGE: java Caesar in out key");
15.        return;
16.      }
17.
18.      try
19.      {
20.        FileInputStream in = new FileInputStream(args[0]);
21.        FileOutputStream out = new FileOutputStream(args[1]);
22.        int key = Integer.parseInt(args[2]);
23.        int ch;
24.        while ((ch = in.read()) != -1)
25.        {
26.          byte c = (byte)(ch + key);
27.          out.write(c);
28.        }
29.        in.close();
30.        out.close();
31.      }
32.      catch (IOException exception)
33.      {
34.        exception.printStackTrace();
35.      }
36.    }
37.  }
```

API java.lang.Class 1.0

- `ClassLoader getClassLoader()`

  Возвращает загрузчик для данного класса.

API java.lang.ClassLoader 1.0

- `ClassLoader getParent()` 1.2

  Возвращает родительский загрузчик классов или значение `null` в случае первичного загрузчика.

- `static ClassLoader getSystemClassLoader()` 1.2

  Возвращает системный загрузчик классов, т.е. тот, который использовался для загрузки первого класса приложения.

- `protected Class findClass(String name)` 1.2

  Данный метод необходимо переопределять в загрузчике классов для поиска байтового кода класса и представления его виртуальной машине путем вызова метода `defineClass()`. В имени класса для отделения имени пакета нужно использовать точку и не указывать суффикс `.class`.

- `Class defineClass(String name, byte[] byteCodeData, int offset, int length)`

  Передает виртуальной машине новый класс, предоставляя байтовый код в определенном диапазоне данных.

API java.net.URLClassLoader 1.2

- `URLClassLoader(URL[] urls)`
- `URLClassLoader(URL[] urls, ClassLoader parent)`

  Создает загрузчик классов, загружающий классы, находящиеся по определенным URL-адресам. Если URL-адрес заканчивается символом /, подразумевается, что речь идет о каталоге, а если нет, то значит — о файле JAR.

API java.lang.Thread 1.0

- `ClassLoader getContextClassLoader()` 1.2

  Возвращает загрузчик классов, который обозначен создателем данного потока, в качестве наиболее приемлемого для использования.

- `void setContextClassLoader(ClassLoader loader)` 1.2

  Задает загрузчик классов для кода в данном потоке. Если при запуске потока контекстный загрузчик классов не задан явно, то используется контекстный загрузчик родителя.

Верификация байтового кода

Когда загрузчик классов представляет виртуальной машине байтовый код класса, этот код сначала исследуется *верификатором* (verifier). Верификатор проверяет все классы за исключением системных, и определяет те команды, которые могут нанести ущерб виртуальной машине.

Верификацию можно отменить с помощью недокументированной опции `-noverify`:

```
java -noverify Hello
```

Ниже приведен перечень некоторых видов проверок, которые производятся верификатором.

- Инициализация переменных перед их использованием.
- Согласование типов ссылок при вызове метода.
- Соблюдение правил доступа к приватным данным и методам.
- Доступ к локальным переменным в рамках стека.
- Отсутствие переполнения стека.

В случае нарушения какого-либо из этих условий класс считается поврежденным и загружаться не будет.



Теорема Гёделя (Gödel) утверждает, что создание алгоритмов, умеющих обрабатывать программные файлы и принимать решение о том, обладают ли программы ввода тем или иным свойством (например, независимостью от переполнения стека), является невозможным. В таком случае возникает вопрос: каким образом верификатор может определять, что в файле класса не соблюдается соответствие типов, используются неинициализированные переменные и переполняется стек? Нет ли здесь противоречия законам логики? Нет, потому что верификатор *не* является алгоритмом принятия решений в смысле теоремы Гёделя. Если верификатор принимает программу, значит, она действительно безопасна. Однако он также может и отклонять инструкции виртуальной машины, даже если те выглядят вполне безопасными. (Наверняка вам уже доводилось сталкиваться с такой проблемой и инициализировать переменную каким-то фиктивным значением из-за того, что компилятор не мог гарантировать ее должную инициализацию.)

Такая строгая верификация предложена по соображениям безопасности. Случайные ошибки, связанные с использованием неинициализированных переменных, могут легко причинить заметный ущерб программе, если их вовремя не определить. Еще важнее то, что в широко открытом мире глобальной сети необходимо обеспечить надежную защиту от злонамеренных попыток недобросовестных программистов, стремящихся проникнуть в чужую систему или нанести ей урон. Например, если бы можно было модифицировать значения в стеке выполняемой программы или изменять значения приватных полей системных объектов, то любая программа могла бы нарушить систему защиты Web-браузера.

Зачем же нужно создавать специальный верификатор для проверки всех этих условий? Ведь компилятор все равно не позволит сгенерировать файл класса, в котором будет использоваться неинициализированная переменная или область приватных полей будет доступна из другого класса. Действительно, файл класса, сгенерированный компилятором языка Java, всегда проходит верификацию. Однако используемый в этих файлах формат байтового кода хорошо документирован, и опытный программист на языке ассемблера и в шестнадцатеричном редакторе может легко вручную создать файл класса с подтвержденными, но опасными командами для виртуальной машины Java. Таким образом, верификатор блокирует злонамеренно измененные классов, а не только проверяет созданные компилятором файлы.

Рассмотрим процесс изменения файла класса вручную. Начнем с программы `VerifierTest.java` (листинг 9.3). Эта простая программа вызывает метод `fun` и выво-

дит результат его выполнения. Она может быть запущена из командной строки или как аплет. Метод fun выполняет сложение чисел 1 и 2.

```
static int fun()
{
   int m;
   int n;
   m = 1;
   n = 2;
   int r = m + n;
   return r;
}
```

В качестве эксперимента попытаемся скомпилировать приведенный ниже вариант этой программы:

```
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, компилятор обнаружит эту проблему и откажется компилировать программу. Для создания недопустимого файла класса запустим программу javap, чтобы выяснить, как компилятор транслирует метод fun:

```
javap -c VerifierTest
```

Приведенная выше команда показывает байтовый код файла класса в мнемонической форме:

```
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 инструкции 3, т.е. замены istore_1 на istore_0, воспользуемся шестнадцатеричным редактором. Таким образом, локальная переменная 0 (m) будет инициализирована дважды, а локальная переменная 1 (n) вообще не будет инициализирована. Чтобы сделать это, необходимо знать шестнадцатеричные значения инструкций. Эти значения приводятся в книге Тима Линдхольма (Tim Lindholm) и Фрэнка Йеллина (Frank Yellin) *The Java Virtual Machine Specification* (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 такое значение можно с помощью шестнадцатеричного редактора. На рис. 9.4 изображен шестнадцатеричный редактор Gnome с загруженным файлом `VerifierTest.class`, в котором выделены байтовые коды метода `fun()`.

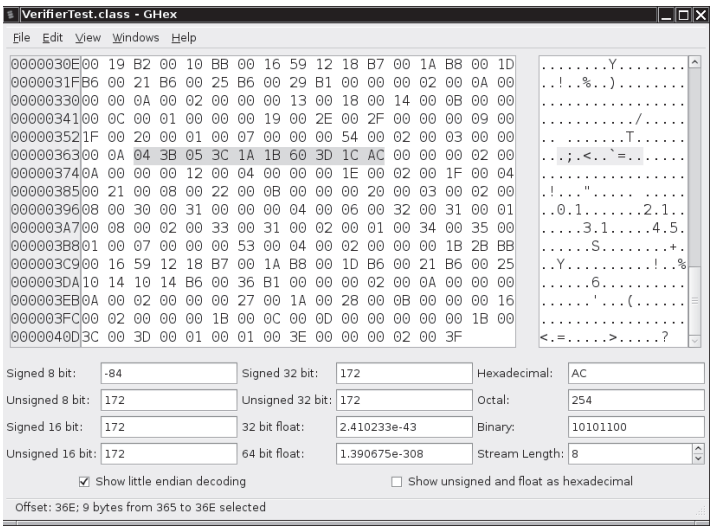


Рис. 9.4. Модификация байтового кода с помощью шестнадцатеричного редактора

Замените значение `3C` значением `3B`, сохраните файл, а затем попытайтесь запустить программу `VerifierTest`. После выполнения этих действий будет получено 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 незаконную модификацию.

Теперь запустим программу, указав опцию `-noverify` (или `-Xverify:none`), т.е. без верификации:

```
java -noverify VerifierTest
```

Метод `fun` возвращает некую случайную величину, получаемую сложением `2` с произвольным значением, которое хранится в переменной `n`. Причина такого поведения в том, что переменная не была инициализирована.

Итак, в результате выполнения программы можно получить следующее:

```
1 + 2 == 15102330
```

Для демонстрации верификации в Web-браузере мы предусмотрели возможность запуска данной программы в виде аплета. Загрузим аплет в Web-браузер, используя следующий URL:

```
file:///c:/CoreJavaBook/v2ch9/VerifierTest/VerifierTest.html
```

Как показано на рис. 9.5, результатом выполнения программы будет сообщение об ошибке, обнаруженной во время верификации.

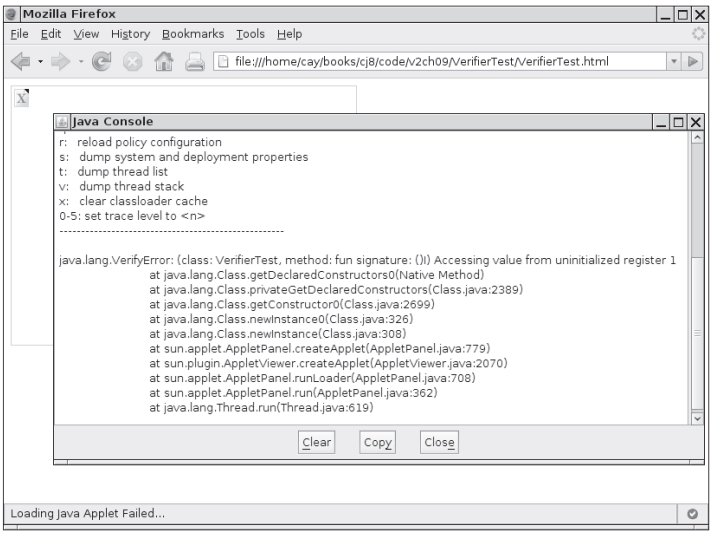


Рис. 9.5. Попытка загрузки преднамеренно измененного файла класса приводит к возникновению ошибки

Листинг 9.3. Содержимое файла `VerifierTest.java`

```
1.  import java.awt.*;
2.  import java.applet.*;
3.
4.  /**
5.   * Пример, демонстрирующий работу верификатора байтового кода виртуальной
6.   * машины. В случае использования шестнадцатеричного редактора для
7.   * изменения файла класса виртуальная машина должна быть способна
8.   * обнаружить подлог.
9.   * @version 1.00 1997-09-10
10.  * @author Cay Horstmann
11.  */
12. public class VerifierTest extends Applet
13. {
14.    public static void main(String[] args)
15.    {
16.       System.out.println("1 + 2 == " + fun());
17.    }
18.
19.    /**
20.     * Метод, вычисляющий 1 + 2.
21.     * @return 3, если код не поврежден
22.     */
23.    public static int fun()
24.    {
25.       int m;
26.       int n;
27.       m = 1;
28.       n = 2;
```

```
29.       // В файле класса заменяем последнее выражение на "m = 2". Для этого
30.       // используем редактор шестнадцатеричных значений.
31.       int r = m + n;
32.       return r;
33.    }
34.
35.    public void paint(Graphics g)
36.    {
37.       g.drawString("1 + 2 == " + fun(), 20, 20);
38.    }
39. }
```

Диспетчеры защиты и полномочия

После загрузки класса в виртуальную машину и прохождения им проверки у верификатора в действие вступает второй механизм обеспечения безопасности платформы Java: *диспетчер защиты* (security manager). Диспетчер защиты — это класс, который следит за тем, разрешено коду выполнять ту или иную операцию или нет. К числу подпадающих под контроль диспетчера защиты операций относятся перечисленные ниже.

- Создание нового загрузчика классов.
- Выход из виртуальной машины.
- Получение классом доступа к члену другого класса.
- Получение доступа к полю другого файла путем применения рефлексии.
- Получение доступа к файлу.
- Открытие соединения через сокет.
- Запуск задания на печать.
- Получение доступа к системному буферу обмена.
- Получение доступа к очереди событий AWT.
- Обращение к окну верхнего уровня.

Существует и много других выполняемых диспетчером защиты проверок, вроде проверок по библиотеке Java.

Поведение по умолчанию при запуске приложений Java не предусматривает установки *никакого* диспетчера защиты, так что все эти операции являются разрешенными. Программа просмотра аплетов, однако, принудительно применяет политику безопасности, которая довольно существенно ограничивает количество разрешенных операций.

Например, аплетам не позволяется завершать работу виртуальной машины. При попытке аплета вызвать метод `exit()`, выдается исключение системы обеспечения безопасности. Более детально это выглядит так. Метод `exit()` класса `Runtime` вызывает метод `checkExit()` диспетчера защиты.

Ниже представлен полный 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 (то есть на завершение работы виртуальной машины): из браузера или из какого-то отдельного аплета. Если диспетчер защиты дает добро на выполнение этого запроса, метод `checkExit()` просто возвращает управление, и дальше процесс обработки продолжается обычным образом. Однако если диспетчер защиты не дает разрешения на выполнение запроса, метод `checkExit()` выдает исключение `SecurityException`.

Метод `exit()` продолжает выполняться только в том случае, если никакого исключения не было. Он затем вызывает *приватный платформенно-ориентированный метод* `exitInternal()`, который уже непосредственно и завершает работу виртуальной машины. Другого способа для завершения работы виртуальной машины не существует, и поскольку метод `exitInternal()` является приватным, ни из какого другого класса он вызываться не может. Следовательно, любой пытающийся выйти из виртуальной машины код должен обязательно вызывать метод `exit()` и, соответственно, проходить осуществляемую с помощью метода `checkExit()` проверку на предмет безопасности без генерации исключения.

Очевидно, что целостность политики обеспечения безопасности зависит от тщательного кодирования. Поставщики системных служб в стандартной библиотеке должны всегда обращаться к диспетчеру защиты перед осуществлением попыток выполнения каких-либо серьезных операций.

Диспетчер защиты платформы Java позволяет как программистам, так и системным администраторам организовывать детальное управление отдельными полномочиями. Более подробно об этом рассказывается уже в следующем разделе. Там сначала приводится краткий обзор предлагаемой в Java 2 модели безопасности, затем демонстрируются способы управления правами доступа с помощью *файлов политики* (policy file), а напоследок объясняется, как можно определять свои собственные типы прав доступа.



Допускается реализовывать и устанавливать свой собственный диспетчер защиты, но это лучше делать только тем, кто является экспертом в области компьютерной безопасности. В принципе гораздо безопаснее просто настраивать нужным образом стандартный механизм диспетчера защиты.

Модель защиты Java

Модель обеспечения безопасности в JDK версии 1.0 была очень проста: локальные классы имели все полномочия, а возможности удаленных классов были ограничены *песочницей* (sandbox), т.е. диспетчер защиты аплета отказывал в любом доступе к локальным ресурсам. Подобно тому, как ребенку разрешают играть в песочнице и строго ограничивают действия за ее пределами, удаленные классы имели возможность лишь выводить информацию на экран и взаимодействовать с пользователем. В версии JDK 1.1 эта модель была немного модифицирована, т.е. заслуживающие доверия (или 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 либо имели полный доступ к ресурсам, либо ограничивались песочницей.

Начиная с версии Java SE 1.2, в платформе Java предлагается гораздо более гибкий механизм защиты. *Политика безопасности* (security policy) отображает *источники кода* (code source) на *наборы полномочий* (permission set), как показано на рис. 9.6.

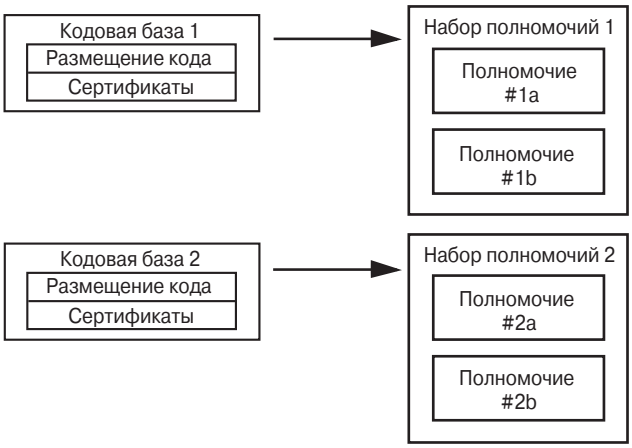


Рис. 9.6. Политика безопасности

Каждый *источник кода* характеризуется *кодовой базой* (code base) и набором *сертификатов*. Кодовая база указывает на место происхождения кода. Например, кодовой базой кода удаленного аплета будет URL-адрес HTTP, с которого этот аплет был загружен, а кодовой базой кода в JAR-файле, соответственно – URL-адрес, с которого был загружен этот файл. Сертификаты, если таковые присутствуют, служат предоставляемой соответствующей компанией гарантией того, что данный код не является подделкой. Более подробно о них будет рассказываться позже в этой главе.

Под *полномочиями*, или *правами доступа*, подразумеваются любые свойства, которые проверяются диспетчером защиты (для обозначения данного понятия еще также может использоваться и термин *привилегии*). Платформа Java поддерживает целый ряд представляющих полномочия классов, каждый из которых инкапсулирует детали конкретного полномочия. Например, ниже показан экземпляр класса `FilePermission`, который разрешает чтение и запись любого файла в каталоге `/tmp`:

```
FilePermission p = new FilePermission("/tmp/*", "read,write");
```

Однако даже еще более важным является то, что в стандартной реализации класс `Policy` считывает информацию о привилегиях из *файла прав доступа* (permissions file). Внутри этого файла та же привилегия на выполнение чтения и записи, что приводилась выше, будет выглядеть так:

```
permission java.io.FilePermission "/tmp*", "read,write";
```

Более подробно о файлах прав доступа будет рассказываться в следующем разделе.

На рис. 9.7 показана иерархия классов полномочий, которые поставлялись вместе Java SE 1.2. В последующих версиях Java добавилось еще множество дополнительных классов полномочий.

В предыдущем разделе было показано, что у класса `SecurityManager` имеются методы проверки обеспечения безопасности наподобие `checkExit()`. Эти методы существуют только для удобства программистов и обратной совместимости. Все они отображаются на стандартные проверки полномочий. Например, исходный код метода `checkExit()` выглядит следующим образом:

```
public void checkExit()
{
   checkPermission(new RuntimePermission("exitVM"));
}
```

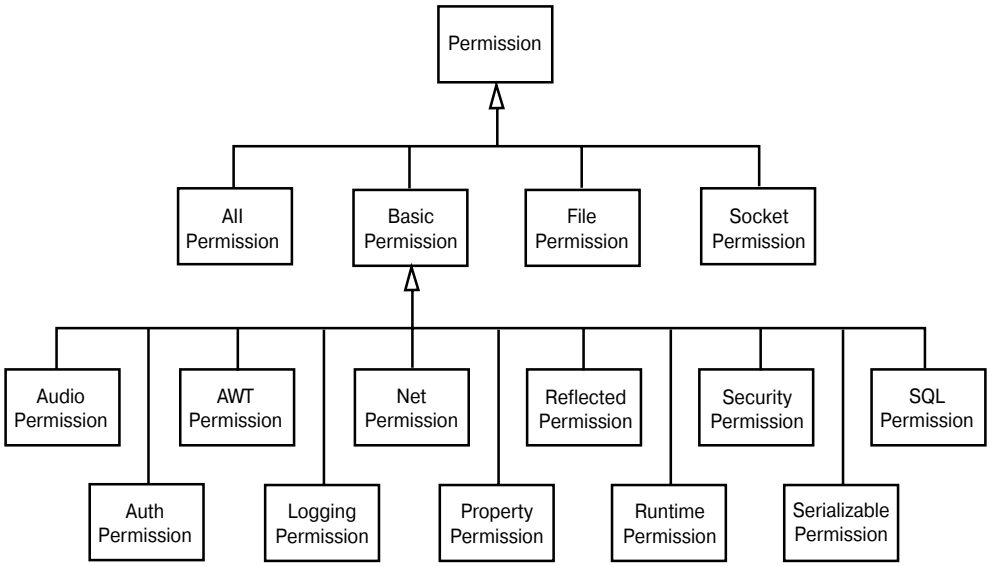


Рис. 9.7. Часть иерархии классов полномочий

У каждого класса имеется *домен обеспечения безопасности*, или *домен защиты* (protection domain), т.е. объект, который инкапсулирует как источник кода, так и набор прав доступа этого класса. При необходимости проверить то или иное полномочие, диспетчер защиты `SecurityManager` сначала выясняет, к каким классам относятся все находящиеся в текущий момент в стеке вызовов методы. Затем он получает доступ к доменам обеспечения безопасности всех этих классов и узнает у них о том, допускает ли их набор полномочий выполнение проверяемой им сейчас операции. Если все домены дают добро на выполнение этой операции, тогда проверка завершается успехом, а если нет — то выдачей исключения `SecurityException`.

Почему выполнение операции должны позволять все методы в стеке вызовов? Давайте рассмотрим пример. Предположим, что метод `init()` аплета собирается открыть файл с помощью следующего вызова:

```
Reader in = new FileReader(name);
```

Конструктор `FileReader` вызывает конструктор `FileInputStream`, который, в свою очередь, обращается к методу `checkRead()` диспетчера защиты, а тот затем уже, наконец, вызывает метод `checkPermission()` с объектом `FilePermission(name, "read")`. Внешний вид стека вызовов в таком случае показан в табл. 9.1.

Таблица 9.1. Стек вызовов во время проверки полномочий

Класс	Метод	Источник кода	Права доступа
`SecurityManager`	`checkPermission`	`null`	`AllPermission`
`SecurityManager`	`checkRead`	`null`	`AllPermission`
`FileInputStream`	Конструктор	`null`	`AllPermission`
`FileReader`	Конструктор	`null`	`AllPermission`
Аплет	`init`	Источник кода аплета	Права доступа аплета
...			

Классы `FileInputStream` и `SecurityManager` являются *системными классами*, источник кода у которых выглядит как `null`, а набор полномочий — как экземпляр класса `AllPermission`, который разрешает выполнять все операции. Очевидно, что полномочия только этих классов не могут определить исход проверки. Помимо них метод `checkPermission()` должен также обязательно взять в расчет и ограниченные полномочия класса аплета. За счет проверки всего стека вызовов, механизм обеспечения безопасности исключает вероятность выполнения классом важных операций от имени какого-то другого класса.



Приведенное краткое описание процесса проверки полномочий позволяет получить лишь общее представление. Многие технические детали здесь, однако, были опущены. Учитывая тот факт, что в вопросах безопасности вся "соль" как раз состоит именно в деталях, для их изучения мы рекомендуем прочитать книгу Ли Гонга (Li Gong) *Inside Java 2 Platform Security* (Addison-Wesley, 1999 г.), а для более критичного анализа — книгу Гэри Мак-Гроу (Gary McGraw) и Эда Фельтена (Ed Felten) *Securing Java: Getting Down to Business with Mobile Code* (Wiley, 1999 г.). Электронную версию последней можно найти по адресу: `http://www.securingjava.com`.

API java.lang.SecurityManager 1.0

- `void checkPermission(Permission p) 1.2`

  Проверяет, предоставляет ли диспетчер защиты данное полномочие. Если нет, выдается исключение `SecurityException`.

API java.lang.Class 1.0

- `ProtectionDomain getProtectionDomain() 1.2`

  Получает доступ к представляющему данный класс домену обеспечения безопасности или возвращает значение `null`, если этот класс был загружен без домена защиты.

API java.security.ProtectionDomain 1.2

- `ProtectionDomain(CodeSource source, PermissionCollection permissions)`

  Создает домен обеспечения безопасности на основе указанного источника кода и прав доступа.

- `CodeSource getCodeSource()`

  Извлекает информацию об источнике кода из данного домена обеспечения безопасности.

- `boolean implies(Permission p)`

  Возвращает значение `true`, если указанное полномочие является разрешенным в данном домене обеспечения безопасности.

API java.security.CodeSource 1.2

- `Certificate[] getCertificates()`

  Извлекает цепочку сертификатов для ассоциируемых с данным источником кода подписей файлов классов.

- `URL getLocation()`

  Извлекает информацию о кодовой базе ассоциируемых с данным источником кода файлов классов.

Файлы политики безопасности

Диспетчер политик выполняет считывание *файлов политики*, в которых содержатся инструкции по отображению источников кода на полномочия. Ниже приведен пример типичного файла политики:

```
grant codeBase "http://www.horstmann.com/classes"
{
  permission java.io.FilePermission "/tmp/*", "read, write";
}
```

Этот файл предоставляет право на выполнение считывания и записи файлов в каталоге `/tmp` любому коду, который был загружен с `http://www.horstmann.com/classes`.

Устанавливаться файлы политики могут только в стандартных местах. По умолчанию таковыми являются:

- файл `java.policy` в основном каталоге платформы Java;
- файл `.java.policy` (обратите внимание на стоящую в начале его имени точку) в рабочем каталоге пользователя.



Стандартные пути размещения этих файлов могут изменяться путем использования конфигурационного файла `java.security`, находящегося в каталоге *`jre`*`/lib/security`. По умолчанию пути к этим файлам внутри данного файла 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 политик URL-адреса, находящиеся вообще на каком-нибудь другом сервере и исключающие возможность внесения изменений в политики пользователями. URL-адресов политик (с последовательными номерами) в файле политик может быть сколько угодно. Предусматриваемые всеми такими файлами полномочия все равно будут комбинироваться.

При желании сделать так, чтобы политики хранились за пределами файловой системы, можно реализовать подкласс класса `Policy`, собирающий информацию обо всех полномочиях, и затем внутри конфигурационного файла `java.security` просто изменить следующую строку показанным ниже образом:

```
policy.provider=sun.security.provider.PolicyFile
```

Во время тестирования постоянно менять стандартные файлы политик не имеет смысла, поэтому мы предлагаем поместить все полномочия в отдельный файл, скажем, файл `MyApp.policy`, и затем явным образом указывать имя этого файла политик для каждого рассматриваемого приложения. Применять политику можно двумя способами, а именно — либо путем установки соответствующим образом свойства системы внутри главного метода приложений:

```
System.setProperty("java.security.policy", "MyApp.policy");
```

либо путем запуска виртуальной машины с помощью следующей команды:

```
java -Djava.security.policy=MyApp.policy MyApp
```

В случае аплетов вместо этого следует применять такую команду:

```
appletviewer -J-Djava.security.policy=MyApplet.policy MyApplet.html
```

(За счет использования в appletviewer опции -J виртуальной машине может передаваться любой аргумент командной строки.)

В приведенных примерах файл MyApp.policy, по сути, добавляется к остальным политикам. В случае включения второго знака равенства, однако, то есть в случае применения команды java -Djava.security.policy==MyApp.policy MyApp, в приложении будет использоваться *только* данный указанный файл политики, а все стандартные — игнорироваться.



Во время тестирования можно легко допустить ошибку, случайно оставив в текущем каталоге файл .java.policy, который может предоставлять массу, а то и все возможные полномочия (AllPermission). Поэтому, заметив, что приложение как будто не замечает ограничений, указанных в добавленном файле политик, первое, что нужно сделать — проверить, не остался ли в текущем каталоге файл java.policy. Допущение подобной ошибки является наиболее вероятным на системах UNIX, поскольку на таких системах файлы с начинающимися с точки именами по умолчанию не отображаются.

Как уже упоминалось ранее, диспетчер защиты в приложениях Java по умолчанию не устанавливается. Это означает, что до тех пор, пока таковой не будет установлен, увидеть действие файлов политики не получится. Установить диспетчер защиты можно либо путем добавления в метод main() следующей строки:

```
System.setSecurityManager(new SecurityManager());
```

либо указанием при запуске виртуальной машины опции командной строки -Djava.security.manager:

```
java -Djava.security.manager -Djava.security.policy=MyApp.policy MyApp
```

В остальной части этого раздела довольно подробно рассматриваются способы описания полномочий в файле политики с затрагиванием всех касающихся его форматирования деталей, кроме способов указания сертификатов, о которых более подробно будет рассказываться далее в этой главе.

В файле политики содержится последовательность записей grant, каждая из которых имеет следующий вид:

```
grant источник_кода
{
   право_доступа_1;
   право_доступа_2;
   ...
}
```

В источнике кода содержится информация о кодовой базе (которая может опускаться, если данная запись grant распространяется на код из всех источников), а также имена пользующихся доверием принципалов и авторов сертификатов (которые тоже могут опускаться, если подписи не являются обязательными для данной записи).

Информация о кодовой базе задается следующим образом:

```
codeBase "URL"
```

Наличие в конце URL-адреса символа косой черты (“/”) означает, что он ссылается на каталог, а отсутствие — что он ссылается на JAR-файл:

```
grant codeBase "www.horstmann.com/classes/" {...}
grant codeBase "www.horstmann.com/classes/MyApp.jar" {...}
```

В представляющих кодовую базу URL-адресах в качестве разделителя файлов должны всегда использоваться символы левой косой черты, даже если речь идет об URL-адресах в операционной системе Windows. Например:

```
grant codeBase "file:C:/myapps/classes/" { . . . };
```



Всем известно, что URL-адреса типа `http` всегда начинаются двумя символами косой черты (`http://`). Однако в случае URL-адресов типа `file`, похоже, часто возникают излишние сомнения из-за того, что программа чтения файлов политики разрешает использовать для них два формата, а именно — формат `file://`*`локальный_файл`* и формат `file:`*`локальный_файл`*. Более того, последняя косая черта перед именем диска в Windows является не 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 наших тестовых примерах вариант `file:////C:/dir/filename.ext` тоже является приемлемым, однако объяснить этого мы не можем.

Полномочия задаются с помощью следующего синтаксиса:

```
permission имя_класса имя_объекта_доступа, список_действий;
```

На месте *`имя_класса`* указывается полностью уточненное имя представляющего данное полномочие класса (например, `java.io.FilePermission`), а на месте *`имя_объекта_доступа`* — конкретный целевой объект, на который должно распространяться действие данного полномочия; например, в случае касающегося доступа к файлам полномочия это может быть имя файла и каталога, а в случае касающегося доступа к сокетам полномочия — название и номер порта хоста. На месте параметра `список_действий` указывается перечень охватываемых данным полномочием допустимых действий, вроде действий `read` или `connect`, которые отделяются друг от друга запятой. Некоторые классы полномочий не требуют указывать ни имена целевых объектов, ни списки действий. В табл. 9.2 перечислены некоторые наиболее часто применяемые классы полномочий и охватываемые ими действия.

Таблица 9.2. .Классы полномочий и ассоциируемые с ними целевые объекты и действия

Полномочие	Целевые объекты	Действия
`java.io.FilePermission`	Файл	`read`, `write`, `execute`, `delete`
`java.net.SocketPermission`	Сетевое соединение	`accept`, `connect`, `listen`, `resolve`
`java.util.PropertyPermission`	Свойство	`read`, `write`

Продолжение табл. 9.2

Полномочие	Целевые объекты	Действия
java.lang.RuntimePermission	createClassLoader getClassLoader setContextClassLoader enableContextClassLoaderOverride createSecurityManager exitVM getenv.*имя_переменной* shutdownHooks setFactory setIO modifyThread stopThread modifyThreadGroup getProtectionDomain readFileDescriptor writeFileDescriptor loadLibrary.*имя_библиотеки* accessClassInPackage.*имя_пакета* defineClassInPackage.*имя_пакета* accessDeclaredMembers.*имя_класса* queuePrintJob getStackTrace setDefaultUncaughtExceptionHandler preferences usePolicy	отсутствуют
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
jaba.lang.reflect.ReflectPermission	suppressAccessChecks	отсутствуют
java.io.SerializablePermission	enableSubclassImplementation enableSubstitution	отсутствуют

Окончание табл. 9.2

Полномочие	Целевые объекты	Действия
`java.security.SecurityPermission`	`createAccessControlContext` `getDomainCombiner` `getPolicy` `setPolicy` `getProperty.`*`ключевое_имя`* `setProperty.`*`ключевое_имя`* `insertProvider.`*`имя_поставщика`* `removeProvider.`*`имя_поставщика`* `setSystemScope` `setIdentityPublicKey` `setIdentityInfo` `addIdentityCertificate` `removeIdentityCertificate` `printIdentity` `clearProviderProperties.`*`имя_поставщика`* `putProviderProperty.`*`имя_поставщика`* `removeProviderProperty.`*`имя_поставщика`* `getSignerPrivateKey` `setSignerKeyPair`	отсутствуют
`java.security.AllPermission`	отсутствуют	отсутствуют
`javax.audio.AudioPermission`	`play` `record`	отсутствуют
`javax.security.auth.AuthPermission`	`doAs` `doAsPrivileged` `getSubject` `getSubjectFromDomainCombiner` `setReadOnly` `modifyPrincipals` `modifyPublicCredentials` `modifyPrivateCredentials` `refreshCredential` `destroyCredential` `createLoginContext.`*`имя_контекста`* `getLoginConfiguration` `setLoginConfiguration` `refreshLoginConfiguration`	отсутствуют
`java.util.logging.LoggingPermission`	`control`	отсутствуют
`java.sql.SQLPermission`	`setLog`	отсутствуют

Как видно в табл. 9.2, большинство полномочий просто позволяет выполнять определенную операцию. К операции можно относиться как к целевому объекту с подразумеваемым действием "`permit`". Все перечисленные классы полномочий представляют собой подклассы класса `BasicPermission` (см. рис. 9.7). Классы с такими целевыми объектами, как файл, сокет и свойство, однако, являются более сложными и заслуживают более подробного рассмотрения.

Целевые объекты касающихся файлов полномочий могут иметь следующих вид:

файл	Файл
каталог/	Каталог
*каталог/**	Все файлы данного каталога
*	Все файлы текущего каталога
каталог/-	Все файлы данного каталога и всех его подкаталогов
-	Все файлы текущего каталога и всех его подкаталогов
<<ALL FILES>>	Все файлы файловой системы

Например, следующая представляющая полномочие запись означает, что доступ разрешается получать 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 именах пути в операционной системе Windows ее нужно повторять дважды:

```
permission java.io.FilePermission "c:\\myapp\\-",
   "read,write,delete";
```

Целевые объекты касающихся сокетов полномочий требуют указания узла (хоста) и диапазона портов. Синтаксис, необходимый для указания хоста выглядит следующим образом:

имя_хоста или *IP_адрес*	Один узел
`localhost` или пустая строка	Локальный узел
**.суффикс_домена*	Любой узел, принадлежащий домену, имя которого заканчивается указанным суффиксом
*	Все узлы

Номера портов являются необязательными и указываются в соответствии с приведенными ниже соглашениями:

:*n*	Один порт
:*n-*	Все порты с номерами *n* и выше
:*-n*	Все порты с номерами *n* и ниже
:*n1-n2*	Все порты в диапазоне между *n1* и *n2*

Вот пример:

```
permission java.net.SocketPermission
   "*.horstmann.com:8000-8999", "connect";
```

И, наконец, целевые объекты касающихся свойств полномочий могут принимать одну из двух следующих форм:

свойство	Отдельное свойство
*префикс_свойства.**	Все свойства с указанным суффиксом

То есть они могут выглядеть как `"java.home"` и как `"java.vm.*"`.

Например, следующая представляющая полномочие запись означает, что программе разрешается считывать все свойства, которые начинаются с `java.vm`:

```
permission java.util.PropertyPermission "java.vm.*", "read";
```

В файлах политик также могут использоваться и системные свойства. Лексема `${`*свойство*`}` в таком случае заменяется значением свойства.

Например, `${user.home}` заменяется основным каталогом пользователя. Ниже приведен типичный пример использования этого системного свойства в записи, представляющей полномочие:

```
permission java.io.FilePermission "${user.home}", "read,write";
```

Для создания не зависящих от используемой платформы файлов политик вместо явных разделителей / или \\ рекомендуется использовать свойство `file.separator` или даже, чтобы было еще проще, его сокращенную версию `${/}`. Например, для предоставления прав на выполнение чтения и записи в основном каталоге пользователя и всех его подкаталогах, можно использовать такую удобную в плане переносимости запись:

```
permission java.io.FilePermission "${user.home}", "read,write";
```



JDK поставляется с одной примитивной утилитой, которая называется `policytool` и предназначена для редактирования файлов политики (рис. 9.8). Очевидно, что для конечных пользователей эта утилита не подходит, поскольку большинство из предлагаемых в ней параметров будет для них совершенно непонятно. С нашей точки зрения, она больше подходит для системных администраторов, которые предпочитают указывать нужный параметр и щелкать, а не вводить синтаксис самостоятельно. Кроме того, в ней еще также не хватает и понятного набора уровней защиты (наподобие низкого, среднего и высокого уровня), который может быть очень важен для не специалистов. В общем, мы считаем, что платформа Java в принципе содержит все необходимые для создания детальной модели защиты компоненты, но могла бы предусматривать более удобные способы для работы с ними как для конечных пользователей, так и для системных администраторов.

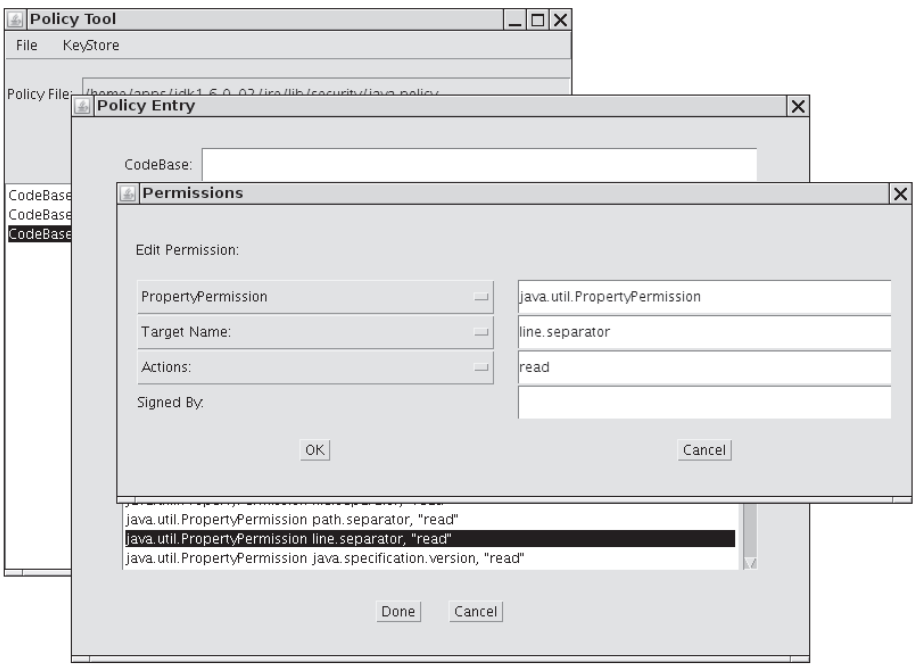


Рис. 9.8. Интерфейс программы `policytool`

Пользовательские права доступа

В этом разделе описывается способ создания собственных классов прав доступа, на которые пользователи могут ссылаться в файлах политики.

Для реализации классов прав доступа 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)`.

Последний метод самый важный. Для прав доступа предусмотрен определенный *порядок* (ordering), в соответствии с которым наиболее общие права *предполагают* (imply) использование специальных прав. Например, показанное ниже право доступа к файлу позволяет чтение и запись любого файла в каталоге `/tmp` и любых его подкаталогах:

```
p1 = new FilePermission("/tmp/-", "read, write");
```

Это общее право предполагает наличие специальных привилегий:

```
p2 = new FilePermission("/tmp/-", "read");
p3 = new FilePermission("/tmp/aFile", "read, write");
p4 = new FilePermission("/tmp/aDirectory/-", " write");
```

Иначе говоря, 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 пример применения метода `implies`. Если конструктор `FileInputStream` открывает файл для чтения, то он проверяет наличие прав такого доступа. Для выполнения этой проверки *специальный* объект прав доступа передается методу `checkPermission()`:

```
checkPermission(new FilePermission(fileName, "read"));
```

После этого диспетчер защиты запрашивает все имеющиеся права доступа о том, предполагается ли специальное право доступа.

В частности, `AllPermission` предполагает все права доступа.

При определении собственных классов прав доступа необходимо определить предполагаемые права доступа.

Например, для управления доступа пользователя Tommy к телевизору под управлением Java в некотором промежутке времени можно определить следующий класс `TVPermission`:

```
new TVPermission("Tommy:2-12:1900-2200", "watch, record")
```

Этот класс позволяет пользователю смотреть и записывать телевизионные передачи на каналах 2–12 в промежутке времени между 19:00 и 22:00. Для применения следующего специального права доступа потребуется реализовать метод `implies()`:

```
new TVPermission("Tommy:4:2000-2100", "watch")
```

Реализация класса полномочий

В этом разделе на примере демонстрационной программы иллюстрируется реализация нового полномочия для осуществления мониторинга за вставляемым в текстовую область текстом. Обязанностью этой программы является предотвращение попыток добавления в текстовую область всевозможных "плохих слов", вроде слова *sex* (секс), *drugs* (наркотики) и *C++*. Для получения возможности передачи списка всех таких плохих слов файлу политики применяется специальный класс полномочий. Он является подклассом класса `JTextArea` и всегда спрашивает у диспетчера защиты о том, можно ли добавлять новый текст, как показано ниже:

```
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);
   }
}
```

Если диспетчер защиты предоставляет полномочие типа `WordCheckPermission`, новый текст добавляется, а если нет, метод `checkPermission()` генерирует исключение.

Полномочия на проверку слов (`WordCheckPermission`) допускают выполнение двух следующих действий: действия `insert()` (подразумевающего осуществление вставки указанного текста) и действия `avoid` (подразумевающего добавление текста без указанных недопустимых слов). Запускать эту программу нужно с помощью следующего файла политики:

```
grant
{
   permission WordCheckPermission "sex,drugs,C++", "avoid";
};
```

Этот файл политики предоставляет разрешение на выполнение вставки любого текста, который не содержит таких недопустимых слов, как *sex*, *drugs* и *C++*.

При создании класса `WordCheckPermission` требуется уделить особое внимание методу `implies()`. Ниже перечислены правила, отвечающие за то, должно ли полномочие `p1` подразумевать полномочие `p2`.

- Если у `p1` имеется действие `avoid`, а у `p2` — действие `insert`, тогда конечный объект `p2` должен исключать все слова из `p1`. Например, полномочие

  ```
  WordCheckPermission "sex,drugs,C++", "avoid"
  ```

  подразумевает полномочие

  ```
  WordCheckPermission "Mary had a little lamb", "insert"
  ```

- Если и у `p1` и у `p2` имеется действие `avoid`, тогда набор слов `p2` должен содержать все слова из набора слова `p1`. Например, полномочие

  ```
  WordCheckPermission "sex,drugs", "avoid"
  ```

  подразумевает полномочие

  ```
  WordCheckPermission "sex,drugs,C++", "avoid"
  ```

- Если и у p1 и у p2 имеется действие insert, тогда текст p1 должен содержать текст из p2. То есть полномочие

  ```
  WordCheckPermission "Mary had a little lamb", "insert"
  ```

  подразумевает полномочие

  ```
  WordCheckPermission "a little lamb", "insert"
  ```

Код реализации этого класса можно найти в листинге 9.4.

Обратите внимание на то, что целевой объект полномочий извлекается с помощью такого имеющего неоднозначное имя метода класса Permission, как getName().

Поскольку полномочия описываются в файлах политик с помощью всего лишь пары строк, класс полномочий должны быть готовы к выполнению синтаксического анализа этих строк. В частности, здесь используется следующий метод для преобразования разделенного запятыми списка ассоциируемых с полномочием avoid недопустимых слов в объект Set:

```
public Set<String> badWordSet()
{
   Set<String> set = new HashSet<String>();
   set.addAll(Arrays.asList(getName().split(",")));
   return set;
}
```

Этот код позволяет использовать для осуществления сравнения наборов слов метод equals() и метод containsAll(). Как рассказывалось в главе 2, метод equals() класса Set признает два набора равными в том случае, если в них содержатся одинаковые элементы, независимо в каком порядке. Например, наборы слов "sex,drugs,C++" и "C++,drugs,sex" будут признаны равными.



Класс полномочий должен обязательно быть классом типа public. Загрузчик файлов политики не может загружать классы с уровнем видимости пакета, выходящим за пределы видимости пути к классам загрузки, и потому просто без всяких предупреждений игнорирует любые классы, которые ему не удается найти.

Код, приведенный в листинге 9.5, показывает, как работает класс WordCheckPermission. Попробуйте ввести в текстовой области какой-нибудь текст и щелкнуть на кнопке Insert (Вставить). Если проверка на предмет безопасности пройдет успешно, текст будет вставлен в текстовую область, а если нет, появится сообщение об ошибке (рис. 9.9).

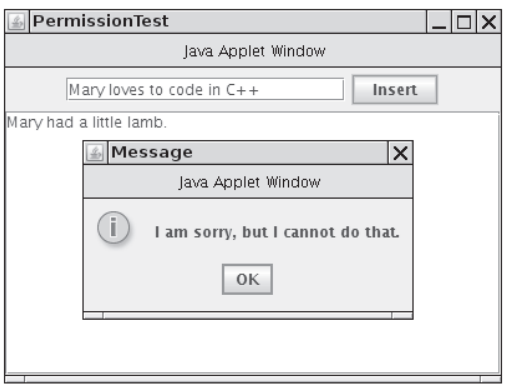


Рис. 9.9. Выполнение программы PermissionTest



При внимательном рассмотрении рис. 9.9 можно заметить, что у окна фрейма имеется граница с не совсем корректным заголовком `Java Applet Window`. За отображение заголовка окна отвечает целевой объект `showWindowWithoutWarningBanner()` класса `java.awt.AWTPermission`. При желании можно отредактировать файл политики, добавить в него такой класс, а затем с его помощью сделать так, чтобы в окне отображался какой-то другой, более подходящий заголовок.

На этом рассмотрение способов настройки полномочий на платформе Java завершается. В большинстве случаев вы будете просто корректировать нужным образом стандартные полномочия, однако при возникновении необходимости в дополнительных возможностях для управления теперь также легко сможете определять и специальные классы полномочий и конфигурировать их тем же образом, что и стандартные.

Листинг 9.4. Содержимое файла `WordCheckPermission.java`

```
1.  import java.security.*;
2.  import java.util.*;
3.
4.  /**
5.   * Полномочие для выполнения проверки на наличие недопустимых слов.
6.   * @version 1.00 1999-10-23
7.   * @author Cay Horstmann
8.   */
9.  public class WordCheckPermission extends Permission
10. {
11.    /**
12.     * Конструктор права доступа с проверкой
13.     * на наличие недопустимых слов.
14.     * @param target Список слов, разделенных запятыми
15.     * @param anAction "insert" или "avoid"
16.     */
17.    public WordCheckPermission(String target, String anAction)
18.    {
19.       super(target);
20.       action = anAction;
21.    }
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 (!action.equals(b.action)) return false;
34.       if (action.equals("insert"))
35.          return getName().equals(b.getName());
36.       else if (action.equals("avoid"))
37.          return badWordSet().equals(b.badWordSet());
38.       else return false;
39.    }
40.
```

```
41.    public int hashCode()
42.    {
43.       return getName().hashCode() + action.hashCode();
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"))
58.             return b.badWordSet().containsAll(badWordSet());
59.          else if (b.action.equals("insert"))
60.          {
61.             for (String badWord : badWordSet())
62.                if (b.getName().indexOf(badWord) >= 0)
63.                   return false;
64.             return true;
65.          }
66.          else return false;
67.       }
68.       else return false;
69.    }
70.
71.    /**
72.     * Возвращает слова, которые в данном
73.     * праве доступа считаются недопустимыми.
74.     * @return Множество недопустимых слов
75.     */
76.    public Set<String> badWordSet()
77.    {
78.       Set<String> set = new HashSet<String>();
79.       set.addAll(Arrays.asList(getName().split(",")));
80.       return set;
81.    }
82.
83.    private String action;
84. }
```

Листинг 9.5. Содержимое файла `PermissionTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import javax.swing.*;
4.
5.  /**
6.   * Пример применения пользовательского класса
7.   * прав доступа WordCheckPermission.
8.   * @version 1.03 2007-10-06
9.   * @author Cay Horstmann
10.  */
```

```
11. public class PermissionTest
12. {
13.    public static void main(String[] args)
14.    {
15.       System.setProperty("java.security.policy", "PermissionTest.policy");
16.       System.setSecurityManager(new SecurityManager());
17.       EventQueue.invokeLater(new Runnable()
18.       {
19.         public void run()
20.         {
21.           JFrame frame = new PermissionTestFrame();
22.           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.           frame.setVisible(true);
24.         }
25.       });
26.    }
27. }
28.
29. /**
30.  * Фрейм, содержащий поле редактирования и текстовую
31.  * область, защищенную от недопустимых слов.
32.  */
33. class PermissionTestFrame extends JFrame
34. {
35.    public PermissionTestFrame()
36.    {
37.       setTitle("PermissionTest");
38.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
39.
40.       textField = new JTextField(20);
41.       JPanel panel = new JPanel();
42.       panel.add(textField);
43.       JButton openButton = new JButton("Insert");
44.       panel.add(openButton);
45.       openButton.addActionListener(new
46.          ActionListener()
47.          {
48.             public void actionPerformed(ActionEvent event)
49.             {
50.                insertWords(textField.getText());
51.             }
52.          });
53.
54.       add(panel, BorderLayout.NORTH);
55.
56.       textArea = new WordCheckTextArea();
57.       add(new JScrollPane(textArea), BorderLayout.CENTER);
58.    }
59.
60.    /**
61.     * Включение текста в текстовую область.
62.     * Если попытка оказалась неудачной,
63.     * отображается диалоговое окно.
64.     * @param words Включаемый текст
65.     */
66.    public void insertWords(String words)
67.    {
```

```
68.       try
69.       {
70.          textArea.append(words + "\n");
71.       }
72.       catch (SecurityException e)
73.       {
74.          JOptionPane.showMessageDialog(this,
75.             "I am sorry, but I cannot do that.");
76.       }
77.    }
78.
79.    private JTextField textField;
80.    private WordCheckTextArea textArea;
81.    private static final int DEFAULT_WIDTH = 400;
82.    private static final int DEFAULT_HEIGHT = 300;
83. }
84.
85. /**
86.  * Текстовая область, в которой метод append
87.  * отклоняет попытки вставить недопустимые слова.
88.  */
89. class WordCheckTextArea extends JTextArea
90. {
91.    public void append(String text)
92.    {
93.       WordCheckPermission p =
94.          new WordCheckPermission(text, "insert");
95.       SecurityManager manager = System.getSecurityManager();
96.       if (manager != null) manager.checkPermission(p);
97.       super.append(text);
98.    }
99. }
```

API java.security.Permission 1.2

- `Permission(String name)`

  Создает класс полномочия с указанными именем целевого объекта.

- `String getName()`

  Возвращает имя целевого объекта данного класса полномочия.

- `boolean implies(Permission other)`

  Проверяет, подразумевает ли данное полномочие другое полномочие. Это происходит в случае, если во втором полномочии описывается более конкретное условие, вытекающее из условия, указанного в первом полномочии.

Аутентификация пользователей

Начиная с версии Java SE 1.4 и выше в состав платформы Java входит служба JAAS (Java Authentication and Authorization Service — служба аутентификации и авторизации Java). Слово Authentication (аутентификация) в названии этой службы указывает на возможность осуществления идентификации пользователей программ, а слово Authorization (авторизация) — на возможность выполнения проверки их полномочий.

Служба JAAS, по сути, представляет собой встраиваемый API-интерфейс, который выделяет приложения Java из конкретных технологий, применяемых для реализа-

ции аутентификации. Помимо прочих, она поддерживает механизмы регистрации в UNIX, механизмы регистрации в NT, механизм аутентификации Kerberos и механизмы аутентификации на основании сертификатов.

После аутентификации пользователя за ним может закрепляться определенный набор полномочий. Ниже приведен пример, в котором пользователю с именем `Harry` назначается специфический набор полномочий, которого больше нет ни у каких других пользователей:

```
grant principal com.sun.security.auth.UnixPrincipal "harry"
{
  permission java.util.PropertyPermission "user.*", "read";
  . . .
};
```

Здесь класс `com.sun.security.auth.UnixPrincipal` выполняет проверку имени запускающего данную программу пользователя UNIX. Метод `getName` этого класса возвращает регистрационное имя UNIX, которое затем просто сравнивается на предмет равенства с именем `harry`.

Для предоставления диспетчеру защиты возможности выполнять проверку такого оператора `grant` применяется класс `LoginContext`. Ниже приведена общая структура кода регистрации:

```
try
{
  System.setSecurityManager(new SecurityManager());
  LoginContext context = new LoginContext("Login1");
    // Определен в конфигурационном файле JAAS.
  context.login();
    // Получить аутентифицированный объект Subject.
  Subject subject = context.getSubject();
  ...
  context.logout();
}
catch (LoginException exception) // Исключение генерируется,
                                 // если попытка регистрации оказалась неудачной.
{
  exception.printStackTrace();
}
```

Теперь `subject` обозначает прошедшего аутентификацию пользователя.

Строковый параметр `"Login1"` в конструкторе `LoginContext` ссылается на запись, имеющую точно такой же вид в конфигурационном файле JAAS. Ниже приведен пример конфигурационного файла:

```
Login1
{
  com.sun.security.auth.module.UnixLoginModule required;
  com.whizzbang.auth.module.RetinaScanModule sufficient;
};

Login2
{
  . . .
};
```

Конечно, в JDK не предусмотрены никакие модули биометрической регистрации.

В пакете com.sun.security.auth.module поставляются только следующие модули:

```
UnixLoginModule
NTLoginModule
Krb5LoginModule
JndiLoginModule
KeyStoreLoginModule
```

Политика безопасности состоит из набора регистрационных модулей, каждый из которых имеет отметку required, sufficient, requisite или optional. Значение этих ключевых слов можно понять из приведенного ниже описания алгоритма.

- Модули выполняются по очереди до тех пор, пока либо выполнение модуля sufficient не завершится успехом, а выполнение модуля requisite — неудачей, либо пока не будет достигнут конец списка модулей.
- Аутентификация проходит успешно в том случае, если либо выполнение всех модулей required и requisite завершается успешно, либо если ни один из них вообще не выполняется, либо если хотя бы один из модулей sufficient или optional успешно завершает свое выполнение.

Процедура регистрации подразумевает прохождение аутентификации *субъектом* (subject), который может иметь несколько *принципалов* (principals). Каждый принципал просто описывает какое-то свойство субъекта, например, его имя пользователя, идентификатор группы или роль. Как было видно в операторе grant, принципалы управляют правами доступа. Объект com.sun.security.auth.UnixPrincipal описывает регистрационное имя UNIX, а объект UnixNumericGroupPrincipal может выполнять проверку на предмет принадлежности группе UNIX.

Конструкция grant может проверять наличие принципала с помощью следующего выражения:

```
grant класс_принципала "имя_принципала"
```

Следовательно, в рассматриваемом примере она может выглядеть так:

```
grant com.sun.security.auth.UnixPrincipal "harry"
```

При регистрации пользователя в системе тогда, в рамках отдельного контекста управления, должен запускаться код, требующий выполнения проверки принципалов. Для запуска нового привилегированного действия PriviligedAction можно использовать статический метод doAs() или doAsPrivileged().

Оба указанных метода выполняют действие путем вызова метода run() объекта, реализующего интерфейс PrivilegedAction, с использованием полномочий принципалов субъекта:

```
PrivilegedAction<T> action = new
  PrivilegedAction()
  {
     public T run()
     {
        // выполнение с использованием полномочий принципалов субъекта
        ...
     }
   };
T result = Subject.doAs(subject, action);
  // или Subject.doAsPrivileged(subject, action, null)
```

При наличии вероятности выдачи после выполнения действий каких-нибудь исключений вместо этого интерфейса лучше реализовать интерфейс Priviliged ExceptionAction.

Методы doAs() и doAsPrivileged() отличаются друг от друга лишь отдельными деталями. Метод doAs() запускается в текущем контексте управления доступом, а метод doAsPrivileged() — в новом контексте. Кроме того, метод doAsPrivileged() позволяет разделять права для регистрационного кода и бизнес-логики. В нашем демонстрационном приложении регистрационный код имеет следующие полномочия:

```
permission javax.security.auth.AuthPermission "createLoginContext.Login1";
permission javax.security.auth.AuthPermission "doAsPrivileged";
```

Прошедший аутентификацию пользователь получает такое полномочие:

```
permission java.util.PropertyPermission "user.*", "read";
```

Если бы вместо doAsPrivileged() мы использовали метод doAs(), регистрационному коду тоже бы требовалось такое полномочие!

Пример программы, код которой приведен в листингах 9.6 и 9.7, демонстрирует, как можно ограничивать права определенных пользователей. Эта программа — AuthTest — осуществляет аутентификацию пользователя, а затем просто выполняет действие, извлекающее системное свойство.

Чтобы этот пример работал, код процедуры регистрации и код действия должны находиться в двух отдельных файлах JAR:

```
javac *java
jar cvf login.jar AuthTest.class
javac SysPropAction.java
jar cvf action.jar SysPropAction.class
```

В файле политики, код которого приведен в листинге 9.8, указано, что пользователь операционной системы UNIX с именем harry обладает правами на чтение всех файлов. Замените имя harry своим учетным именем, и затем выполните следующую команду:

```
java -classpath login.jar:action.jar
   -Djava.security.policy=AuthTest.policy
   -Djava.security.auth.login.config=jaas.config
   AuthTest
```

В листинге 9.12 показана конфигурация процедуры регистрации.

В операционной системе Windows нужно заменить в файлах AuthTest.policy и jaas.config Unix на NT и использовать для разделения JAR-файлов точку с запятой.

```
java -classpath login.jar;action.jar . . .
```

После этого программа AuthTest должна отображать значение свойства user.home. В случае изменения учетного имени в файле AuthTest.policy, однако, должно выдаваться исключение, поскольку у этого пользователя больше не будет требуемого полномочия.



Очень важно *в точности* выполнить все приведенные инструкции, поскольку даже незначительные отклонения могут привести к получению неверной конфигурации.

Листинг 9.6. Содержимое файла AuthTest.java

```
1.  import java.security.*;
2.  import javax.security.auth.*;
3.  import javax.security.auth.login.*;
4.
5.  /**
```

```
6.    * Данная программа аутентифицирует пользователя, а затем
7.    * выполняет SysPropAction с привилегиями данного пользователя.
8.    * @version 1.01 2007-10-06
9.    * @author Cay Horstmann
10.   */
11.  public class AuthTest
12.  {
13.    public static void main(final String[] args)
14.    {
15.      System.setSecurityManager(new SecurityManager());
16.      try
17.      {
18.        LoginContext context = new LoginContext("Login1");
19.        context.login();
20.        System.out.println("Authentication successful.");
21.                        // Аутентификация прошла успешно
22.        Subject subject = context.getSubject();
23.        System.out.println("subject=" + subject);
24.        PrivilegedAction<String> action = new SysPropAction("user.home");
25.        String result = Subject.doAsPrivileged(subject, action, null);
26.        System.out.println(result);
27.        context.logout();
28.      }
29.      catch (LoginException e)
30.      {
31.        e.printStackTrace();
32.      }
33.    }
34.  }
```

Листинг 9.7. Содержимое файла `SysPropAction.java`

```
1.   import java.security.*;
2.
3.   /**
4.    * Данное действие извлекает системное свойство.
5.    * @version 1.01 2007-10-06
6.    * @author Cay Horstmann
7.    */
8.   public class SysPropAction implements PrivilegedAction
9.   {
10.    /**
11.     * Создание действия для поиска требуемого свойства.
12.     * @param propertyName Имя свойства (например, "user.home")
13.     */
14.    public SysPropAction(String propertyName)
15.       { this.propertyName = propertyName; }
16.
17.    public String run()
18.    {
19.       return System.getProperty(propertyName);
20.    }
21.
22.    private String propertyName;
23.  }
```

Листинг 9.8. Содержимое файла `AuthTest.policy`

```
1.  grant codebase "file:login.jar"
2.  {
3.    permission javax.security.auth.AuthPermission
4.       "createLoginContext.Login1";
5.    permission javax.security.auth.AuthPermission "doAsPrivileged";
6.  };
7.
8.  grant principal com.sun.security.auth.UnixPrincipal "harry"
9.  {
10.   permission java.util.PropertyPermission "user.*", "read";
11. };
```

API javax.security.auth.login.LoginContext 1.4

- `LoginContext(String name)`

  Создает контекст регистрации. Параметр `name` соответствует дескриптору регистрации в конфигурационном файле JAAS.

- `void login()`

  Регистрирует субъект, а в случае неудачной регистрации генерирует исключение `LoginException`. Вызывает метод `login()` для диспетчеров, указанных в конфигурационном файле JAAS.

- `void logout()`

  Отменяет регистрацию субъекта. Вызывает метод `logout()` для диспетчеров, указанных в конфигурационном файле JAAS.

- `Subject getSubject()`

  Возвращает аутентифицированного субъекта.

API javax.security.auth.Subject 1.4

- `Set<Principal> getPrincipals()`

  Возвращает набор принципалов для 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)`

  Выполняет привилегированное действие от имени субъекта. Возвращает объект, сгенерированный методом `run()` класса `PrivilegedAction`. Метод `doAsPrivileged()` выполняет действие в указанном контексте управления доступом. Для него можно указать контекст, полученный ранее путем вызова метода `AccessController.getContext()`. Если задано значение `null`, код будет выполняться в новом контексте.

API java.security.PrivilegedAction 1.4

- `Object run()`

  Данный метод нужно реализовать для выполнения кода от имени субъекта.

API java.security.PrivilegedExceptionAction 1.4

- `Object run()`

  Данный метод нужно реализовать для выполнения кода от имени субъекта. Он может генерировать любые обрабатываемые исключения.

API java.security.Principal 1.1

- `String getName()`

  Возвращает имя данного принципала.

Регистрационные модули JAAS

В этом разделе рассматривается пример использования JAAS, демонстрирующий следующие вещи:

- как можно реализовывать свой собственный модуль регистрации;
- как можно реализовывать аутентификацию *на основе ролей*.

Предоставление собственного модуля регистрации бывает полезным в случаях, когда регистрационная информация сохраняется в базе данных. Даже если вас вполне устраивает стандартный модуль, изучение процесса реализации специального модуля поможет лучше разобраться с предназначением параметров в конфигурационном файле JAAS.

Реализация механизма аутентификации на основе ролей играет важную роль в тех случаях, когда требуется управлять большим количеством пользователей. Помещать имена всех допустимых пользователей в файл политики не практично. Вместо этого лучше делать так, чтобы модуль сопоставлял пользователей с ролями вроде `"admin"` или `"HR"` и предоставлял им права на их основе.

Одной из задач регистрационного модуля является заполнение набора принципалов аутентифицируемого субъекта. Если регистрационный модуль поддерживает роли, он должен также добавлять и объекты `Principal`, описывающие эти роли. В библиотеке Java необходимого для осуществления этой задачи класса нет, поэтому мы написали свой собственный (листинг 9.9). Этот класс просто хранит пары *описание=значение*, вроде `role=admin`. А его метод `getName()` возвращает их, что дает нам возможность добавления в файл политик полномочий на основе ролей:

```
grant principal SimplePrincipal "role=admin" { . . . }
```

Наш регистрационный модуль предусматривает выполнение поиска имен, паролей и ролей пользователей в текстовом файле, содержащем приблизительно такие строки:

```
harry|wombat|admin
carl|mockturtle|HR
```

Конечно, в случае реального регистрационного модуля эта информация хранилась бы в какой-нибудь базе данных или каталоге.

Весь код нашего демонстрационного модуля SimpleLoginModule можно найти в листинге 9.10. Метод checkLogin() проверяет, соответствует ли имя пользователя и пароль какой-то записи в файле паролей. Если соответствует, в набор принципалов данного субъекта добавляются два объекта SimplePrincipal:

```
Set<Principal> principals = subject.getPrincipals();
principals.add(new SimplePrincipal("username", username));
principals.add(new SimplePrincipal("role", role));
```

Остальная часть модуля SimpleLoginModule выглядит довольно просто. Метод initialize() получает перечисленные ниже параметры.

- Подвергаемый аутентификации объект Subject.
- Обработчик для извлечения регистрационной информации.
- Карта sharedState, которая может использоваться для обмена данными между регистрационными модулями.
- Карта options, которая содержит пары имя/значение, задаваемые при конфигурировании модуля регистрации.

Например, предположим, что мы конфигурируем наш модуль следующим образом:

```
SimpleLoginModule required pwfile="password.txt";
```

Тогда этот модуль будет извлекать из карты options параметры pwfile.

Сбором имени пользователя и пароля регистрационный модуль тоже не занимается; за это отвечает отдельный обработчик. Такое разделение обязанностей позволяет использовать один и тоже же модуль, не волнуясь о том, откуда поступает регистрационная информация — из диалогового окна графического пользовательского интерфейса, командной строки консоли или конфигурационного файла.

Этот обработчик задается при создании экземпляра LoginContext, например:

```
LoginContext context = new LoginContext("Login1",
   new com.sun.security.auth.callback.DialogCallbackHandler());
```

DialogCallbackHandler предусматривает отображение для запрашивания имени пользователя и пароля простого диалогового окна с графическим пользовательским интерфейсом, а com.sun.security.auth.callback.TextCallbackHandler позволяет извлекать эту информацию из консоли.

Однако мы в нашем приложении используем для получения имени пользователя и пароля свой собственный графический интерфейс (рис. 9.10) и поэтому создаем специальный обработчик, способный просто сохранять и возвращать эту информацию (листинг 9.11).

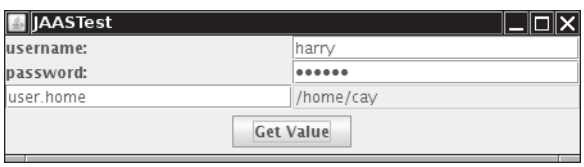


Рис. 9.10. Специальный регистрационный модуль

Этот обработчик состоит из одного единственного метода handle, который обрабатывает массив объектов Callback. Интерфейс Callback реализуется с помощью ряда предопределенных классов, вроде класса NameCallback и класса PasswordCallback. При желании вы можете также добавить и какой-нибудь собственный класс, например, класс RetinaScanCallback.

Код обработчика выглядит немного коряво, поскольку нуждается в выполнении анализа объектов типа Callback.

```
public void handle(Callback[] callbacks)
{
  for (Callback callback : callbacks)
  {
    if (callback instanceof NameCallback) . . .
    else if (callback instanceof PasswordCallback) . . .
    else . . .
  }
}
```

Регистрационный модуль подготавливает массив необходимых ему для проведения аутентификации объектов Callback:

```
NameCallback nameCall = new NameCallback("username: ");
PasswordCallback passCall =
  new PasswordCallback("password: ", false);
callbackHandler.handle(new Callback[] { nameCall, passCall });
```

А затем просто извлекает из этих объектов всю нужную информацию.

При выполнении программы, код которой приведен в листинге 9.12, отображается форма для ввода регистрационной информации и имени системного свойства. В случае успешной аутентификации пользователя, значение этого свойства извлекается в PrivilegedAction. Как видно в файле политики, содержимое которого приведено в листинге 9.13, правом на считывание свойств обладают только пользователи с ролью admin.

Как и в примере из предыдущего раздела, код регистрации и код действия требуется разделить, для чего предлагается создать два следующих JAR-файла:

```
javac *.java
jar cvf login.jar JAAS*.class Simple*.class
jar cvf action.jar SysPropAction.class
```

После этого для запуска программы можно использовать следующую команду:

```
java -classpath login.jar:action.jar
  -Djava.security.policy=JAASTest.policy
  -Djava.security.auth.login.config=jaas.config
  JAASTest
```

В листинге 9.14 приведено содержимое конфигурационного файла.



Еще можно обеспечивать поддержку более сложного двухэтапного протокола и с его помощью делать так, чтобы регистрация считалась *оконченной* только в случае успешного выполнения всех предусмотренных в конфигурации модулей. Более подробная информация об этом доступна в руководстве для разработчиков регистрационных модулей, которое можно найти по адресу http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASLMDevGuide.html.

Листинг 9.9. Содержимое файла SimplePrincipal.java

```
1.  import java.security.*;
2.
3.  /**
4.   * Принципал с именованным значением
5.   * (например, "role=HR" или "username=harry").
6.   */
```

```
7.  public class SimplePrincipal implements Principal
8.  {
9.     /**
10.     * Создание объекта SimplePrincipal, содержащего
11.     * описание и значение.
12.     * @param roleName Имя роли
13.     */
14.    public SimplePrincipal(String descr, String value)
15.       {
16.          this.descr = descr;
17.          this.value = value; }
18.
19.      /**
20.       * Возвращает имя роли для данного принципала.
21.       * @return Имя роли
22.       */
23.    public String getName()
24.    {
25.       return descr + "=" + value;
26.     }
27.
28.    public boolean equals(Object otherObject)
29.    {
30.       if (this == otherObject) return true;
31.       if (otherObject == null) return false;
32.       if (getClass() != otherObject.getClass()) return false;
33.       SimplePrincipal other = (SimplePrincipal) otherObject;
34.       return getName().equals(other.getName());
35.    }
36.    public int hashCode()
37.    {
38.      return getName().hashCode();
39.     }
40.
41.    private String descr;
42.    private String value;
43. }
```

Листинг 9.10. Содержимое файла `SimpleLoginModule.java`

```
1.  import java.io.*;
2.  import java.security.*;
3.  import java.util.*;
4.  import javax.security.auth.*;
5.  import javax.security.auth.callback.*;
6.  import javax.security.auth.login.*;
7.  import javax.security.auth.spi.*;
8.
9.  /**
10.  * Регистрационный модуль, выполняющий аутентификацию
11.  * пользователей по регистрационным именам, паролям
12.  * и ролям, прочитанным из текстового файла.
13.  * @version 1.0 2004-09-14
14.  * @author Cay Horstmann
15.  */
```

```
16. public class SimpleLoginModule implements LoginModule
17. {
18.    public void initialize(Subject subject,CallbackHandler callbackHandler,
19.        Map<String, ?> sharedState, Map<String, ?> options)
20.    {
21.       this.subject = subject;
22.       this.callbackHandler = callbackHandler;
23.       this.sharedState = sharedState;
24.       this.options = options;
25.    }
26.
27.    public boolean login() throws LoginException
28.    {
29.       if (callbackHandler == null) throw new LoginException("no handler");
30.
31.       NameCallback nameCall = new NameCallback("username: ");
32.       PasswordCallback passCall = new PasswordCallback("password: ", false);
33.       try
34.       {
35.          callbackHandler.handle(new Callback[]{ nameCall, passCall });
36.       }
37.       catch (UnsupportedCallbackException e)
38.       {
39.          LoginException e2 = new LoginException("Unsupported callback");
40.          e2.initCause(e);
41.          throw e2;
42.       }
43.       catch (IOException e)
44.       {
45.          LoginException e2 = new LoginException("I/O exception in callback");
46.          e2.initCause(e);
47.          throw e2;
48.       }
49.
50.       return checkLogin(nameCall.getName(), passCall.getPassword());
51.    }
52.
53.    /**
54.     * Проверка корректности аутентификационной информации.
55.     * Если результаты проверки положительны, субъект получает
56.     * принципал, соответствующий пользовательскому имени
57.     * и роли.
58.     * @param username Пользовательское имя
59.     * @param password Символьный массив, содержащий пароль
60.     * @return true если аутентификационная информация корректна
61.     */
62.    private boolean checkLogin(String username, char[] password)
63.             throws LoginException
64.    {
65.       try
66.       {
67.          Scanner in = new Scanner(new FileReader("" +
68.             options.get("pwfile")));
69.          while (in.hasNextLine())
70.          {
71.             String[] inputs = in.nextLine().split("\\|");
```

```
            if (inputs[0].equals(username) &&
               Arrays.equals(inputs[1].toCharArray(), password))
            {
              String role = inputs[2];
              Set<Principal> principals = subject.getPrincipals();
              principals.add(new SimplePrincipal("username", username));
              principals.add(new SimplePrincipal("role", role));
              return true;
            }
          }
          in.close();
          return false;
        }
        catch (IOException e)
        {
          LoginException e2 =
            new LoginException("Can't open password file");
          e2.initCause(e);
          throw e2;
        }
    }

    public boolean logout()
    {
      return true;
    }
    public boolean abort()
    {
       return true;
     }
    public boolean commit()
    {
      return true;
     }

    private Subject subject;
    private CallbackHandler callbackHandler;
    private Map<String, ?> options;
}
```

Листинг 9.11. Содержимое файла `SimpleCallbackHandler.java`

```
import javax.security.auth.callback.*;

/**
 * Простой обработчик, представляющий
 * пользовательское имя и пароль.
 * @version 1.0 2004-09-14
 * @author Cay Horstmann
 */
public class SimpleCallbackHandler implements CallbackHandler
{
   /**
    * Создание обработчика.
    * @param username Пользовательское имя
    * @param password Символьный массив, содержащий пароль
    */
```

```
16.    public SimpleCallbackHandler(String username, char[] password)
17.    {
18.       this.username = username;
19.       this.password = password;
20.    }
21.
22.    public void handle(Callback[] callbacks)
23.    {
24.       for (Callback callback : callbacks)
25.       {
26.          if (callback instanceof NameCallback)
27.          {
28.             ((NameCallback) callback).setName(username);
29.          }
30.          else if (callback instanceof PasswordCallback)
31.          {
32.             ((PasswordCallback) callback).setPassword(password);
33.          }
34.       }
35.    }
36.
37.    private String username;
38.    private char[] password;
39. }
```

Листинг 9.12. Содержимое файла `JAASTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import javax.security.auth.*;
4.  import javax.security.auth.login.*;
5.  import javax.swing.*;
6.
7.  /**
8.   * Программа аутентифицирует пользователя, используя
9.   * специальный модуль регистрации, а затем выполняет
10.  * SysPropAction с привилегиями зарегистрированного
11.  * пользователя.
12.  * @version 1.0 2004-09-14
13.  * @author Cay Horstmann
14.  */
15. public class JAASTest
16. {
17.    public static void main(final String[] args)
18.    {
19.       System.setSecurityManager(new SecurityManager());
20.       EventQueue.invokeLater(new Runnable()
21.        {
22.         public void run()
23.        {
24.          JFrame frame = new JAASFrame();
25.          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26.          frame.setVisible(true);
27.        }
28.      });
29.   }
30. }
```

```
31. /**
32.  * Фрейм, содержащий поля редактирования для ввода
33.  * пользовательского имени, пароля и имени свойства,
34.  * а также поле для отображения значения свойства.
35.  */
36. class JAASFrame extends JFrame
37. {
38.    public JAASFrame()
39.    {
40.       setTitle("JAASTest");
41.
42.       username = new JTextField(20);
43.       password = new JPasswordField(20);
44.       propertyName = new JTextField(20);
45.       propertyValue = new JTextField(20);
46.       propertyValue.setEditable(false);
47.
48.       JPanel panel = new JPanel();
49.       panel.setLayout(new GridLayout(0, 2));
50.       panel.add(new JLabel("username:"));
51.       panel.add(username);
52.       panel.add(new JLabel("password:"));
53.       panel.add(password);
54.       panel.add(propertyName);
55.       panel.add(propertyValue);
56.       add(panel, BorderLayout.CENTER);
57.
58.       JButton getValueButton = new JButton("Get Value");
59.       getValueButton.addActionListener(new ActionListener()
60.          {
61.             public void actionPerformed(ActionEvent event)
62.                {
63.                  getValue();
64.                 }
65.          });
66.       JPanel buttonPanel = new JPanel();
67.       buttonPanel.add(getValueButton);
68.       add(buttonPanel, BorderLayout.SOUTH);
69.       pack();
70.    }
71.
72.    public void getValue()
73.    {
74.       try
75.       {
76.          LoginContext context = new LoginContext("Login1",
77.             new SimpleCallbackHandler(username.getText(),
78.                password.getPassword()));
79.          context.login();
80.          Subject subject = context.getSubject();
81.          propertyValue.setText(""
82.             + Subject.doAsPrivileged(subject,
83.             new SysPropAction(propertyName.getText()), null));
84.          context.logout();
85.       }
86.       catch (LoginException e )
87.       {
```

```
88.          JOptionPane.showMessageDialog(this, e);
89.       }
90.    }
91.    private JTextField username;
92.    private JPasswordField password;
93.    private JTextField propertyName;
94.    private JTextField propertyValue;
95. }
```

Листинг 9.13. Содержимое файла `JAASTest.policy`

```
1.  grant codebase "file:login.jar"
2.  {
3.    permission java.awt.AWTPermission "showWindowWithoutWarningBanner";
4.    permission javax.security.auth.AuthPermission "createLoginContext.Login1";
5.    permission javax.security.auth.AuthPermission "doAsPrivileged";
6.    permission javax.security.auth.AuthPermission "modifyPrincipals";
7.    permission java.io.FilePermission "password.txt", "read";
8.  };
9.  grant principal SimplePrincipal "role=admin"
10. {
11.   permission java.util.PropertyPermission "*", "read";
12. };
```

Листинг 9.14. Содержимое файла `jaas.config`

```
1.  Login1
2.  {
3.    SimpleLoginModule required pwfile="password.txt";
4.  };
```

API `javax.security.auth.callback.CallbackHandler 1.4`

- `void handle(Callback[] callbacks)`

  Обрабатывает указанные объекты `Callback`, при необходимости взаимодействуя с пользователем, и сохраняет в них требуемую для безопасности информацию.

API `javax.security.auth.callback.NameCallback 1.4`

- `NameCallback(String prompt)`
- `NameCallback(String prompt, String defaultName)`

  Создает объект `NameCallback` с указанным приглашением и именем по умолчанию.

- `void setName(String name)`
- `String getName()`

  Устанавливает или извлекает имя, полученное посредством текущего объекта `Callback`.

- `String getPrompt()`

  Извлекает приглашение, которое должно использоваться при запрашивании данного имени.

- `String getDefaultName()`

  Извлекает имя по умолчанию, которое должно использоваться при запрашивании данного имени.

API javax.security.auth.callback.PasswordCallback 1.4

- `PasswordCallback(String prompt, boolean echoOn)`

  Создает объект `PasswordCallback` с указанным приглашением и флагом режима эхо.

- `void setPassword(char[] password)`
- `char[] getPassword()`

  Устанавливает или извлекает пароль, полученный посредством текущего объекта `Callback`.

- `String getPrompt()`

  Извлекает приглашение, которое должно использоваться при запрашивании данного пароля.

- `boolean isEchoOn()`

  Возвращает отвечающий за режим эхо флаг, который должен использоваться при запрашивании данного пароля.

API javax.security.auth.spi.LoginModule 1.4

- `void initialize(Subject subject, CallbackHandler handler, Map<String,?> sharedState, Map<String,?> options)`

  Инициализирует объект `LoginModule` для аутентификации указанного субъекта. Для получения регистрационной информации использует заданный обработчик. Карта `sharedState` применяется для взаимодействия с другими регистрационными модулями. Карта `options` содержит имена и значения, указанные при конфигурировании данного экземпляра модуля.

- `boolean login()`

  Обеспечивает выполнение процесса регистрации и заполняет принципалы субъекта. Возвращает значение `true`, если регистрация прошла успешно.

- `boolean commit()`

  Если сценарий предполагает двухэтапную процедуру, данный метод вызывается после успешного завершения всех регистрационных модулей. Возвращает значение `true` при успешном выполнении операции.

- `boolean abort()`

  Вызывается, если неудача выполнения другого модуля предполагает отказ от регистрации. Возвращает `true`, если операция выполнена успешно.

- `boolean logout()`

  Отменяет регистрацию субъекта. Возвращает `true`, если операция выполнена успешно.

Цифровые подписи

Как уже упоминалось ранее, именно применение аплетов вызвало наибольший интерес к платформе Java. Программисты сразу поняли, что с практической точки зрения, несмотря на широкие анимационные возможности, аплеты не полностью поддерживают модель безопасности. В пакете JDK 1.0 аплеты очень строго контролируются.

С другой стороны, в защищенной корпоративной сети компании риск атаки посредством аплетов минимален, поэтому логично было бы предоставить аплетам, выполняющимся в ней, дополнительные права. Специалисты компании Sun быстро осознали преимущества, связанные с применением аплетов, и решили предоставить пользователям возможность присваивать аплетам *разные* уровни защиты в зависимости от их происхождения. Например, если аплет поступил от надежного поставщика, которому вы доверяете, такому аплету можно присвоить более обширные права доступа.

Для предоставления аплету дополнительных полномочий необходимо знать две вещи.

1. Откуда поступил аплет?
2. Не был его код поврежден во время передачи?

За последние 50 лет специалисты в области математики и информатики разработали много сложных алгоритмов поддержки целостности данных и цифровых подписей. Многие из них реализованы в пакете `java.security`. Для использования таких алгоритмов совсем не обязательно понимать математические принципы, на которых они основаны. В следующих разделах описывается механизм, с помощью которого *дайджест сообщения* (message digest) позволяет обнаружить факт изменения данного документа, а также способы, посредством которых цифровая подпись подтверждает личность пользователя.

Дайджест сообщения

Дайджест сообщения — это цифровой "отпечаток" блока данных. Например, хеш-алгоритм обеспечения безопасности SHA1 (Secure Hash Algorithm #1) ставит в соответствие блоку данных произвольного размера 160 бит (20 байт). По аналогии с отпечатками пальцев, считается, что не существует двух одинаковых отпечатков SHA1. На самом деле это не так, поскольку алгоритм SHA1 поддерживает 2^{160} отпечатков, поэтому теоретически они могут совпадать. Но число 2^{160} настолько велико, что вероятность дублирования очень мала. Согласно книге Джеймса Уолша (James Walsh) *True Odds: How risks Affect Your Everyday Life* (Merritt Publishing, 1996 г.), вероятность смерти от удара молнии составляет 1/30 000. Если теперь подсчитать вероятность такого же исхода для *10 человек* (выбранных, например, злобным врагом), она окажется гораздо выше, чем вероятность наличия двух одинаковых отпечатков SHA1. (Конечно, от удара молнии погибло гораздо больше, чем 10 человек, но здесь речь идет о *специально выбранной группе* людей.)

Дайджест сообщения имеет два важных свойства.

- Если изменяется один или несколько битов данных, то дайджест сообщения тоже будет изменен.
- Нельзя изменить оригинальное сообщение таким образом, чтобы полученное поддельное сообщение имело такой же дайджест, что и у оригинального сообщения.

Второе свойство, конечно, соблюдается с определенной степенью вероятности. Допустим, что некий миллиардер составил завещание.

```
Upon my death, my property shall be divided equally among my
children; however, my son George shall receive nothing.

(После смерти мое имущество должно быть разделено между моими
детьми; однако мой сын Джордж ничего не получит.)
```

Отпечаток SHA1 для данного сообщения имеет такой вид:

```
2D 8B 35 F3 BF 49 CD B1 94 04 E0 66 21 2B 5E 57 70 49 E1 7E
```

Предположим, что миллиардер отдал текст завещания одному адвокату, а его отпечаток — другому. Далее допустим, что Джордж подкупил адвоката, у которого хранится завещание, и слово `George` заменил словом `Bill`. В этом случае цифровой отпечаток поддельного завещания будет отличаться от отпечатка оригинального завещания:

```
2A 33 0B 4B B3 FE CC 1C 9D 5C 01 A7 09 51 0B 49 AC 8F 98 92
```

Может ли Джордж составить такое поддельное завещание, которое имело бы отпечаток оригинального завещания? Нет, потому что для перебора всех вариантов ему не хватило бы времени существования Земли, даже если бы он был счастливым обладателем миллиарда компьютеров, которые могут перебирать миллион вариантов завещания в секунду.

Для вычисления таких дайджестов сообщений было разработано несколько алгоритмов. Двумя наиболее известными из них являются: алгоритм SHA1, который был разработан Национальным институтом стандартов и технологий (National Institute of Standards and Technology — NIST), и алгоритм MD5, который был изобретен Рональдом Райвестом (Ronald Rivest) из Массачусетского технологического института (Massachusetts Institute of Technologies — MIT). Оба этих алгоритма умеют шифровать фрагменты сообщений своим оригинальным способом. Подробнее о них можно прочитать, к примеру, в книге Вильяма Столлингса (William Stallings) *Cryptography and Network Security, 4th Edition* (Prentice Hall, 2005 г.). Стоит отметить, однако, что недавно в обоих этих алгоритмах были обнаружены незначительные изъяны. В настоящее время многие специалисты по шифрованию рекомендуют избегать алгоритма MD5 и пользоваться алгоритмом SHA1 до тех пор, пока не появится какой-то более мощный альтернативный вариант. (Более подробную информацию об этом можно найти по адресу `http://www.rsa.com/rsalabs/node.asp?id=2834`.)

В языке Java реализованы оба эти алгоритма. Класс `MessageDigest` представляет собой *фабрику* (factory) объектов, которые реализуют алгоритмы создания цифровых отпечатков. Данный класс содержит статический метод `getInstance()`, который возвращает экземпляр подкласса класса `MessageDigest`. Поэтому класс `MessageDigest` может выступать в роли класса-фабрики или суперкласса для всех алгоритмов создания профиля сообщения.

Ниже приведен пример получения объекта для расчета отпечатка согласно алгоритму SHA1:

```
MessageDigest alg = MessageDigest.getInstance("SHA-1");
```

(Для получения объекта, который использует алгоритм MD5, нужно в качестве аргумента метода `getInstance` вместо строки `"SHA-1"` указать строку `"MD5"`.)

После создания объекта `MessageDigest` ему необходимо передать все байты сообщения с помощью метода `update()`. Например, приведенный ниже фрагмент кода передает содержимое файла созданному выше объекту `alg`, формирующему цифровой отпечаток:

```
InputStream in = . . .
int ch;
while ((ch = in.read()) != -1)
   alg.update((byte) ch);
```

Если байты размещены в массиве, то метод `update()` можно применить для всего массива:

```
byte[] bytes = ...;
alg.update(bytes);
```

Далее следует вызвать метод `digest()`, который дополняет входной поток недостающими битами (необходимое условие использования алгоритма создания отпечатков), рассчитывает цифровой отпечаток и возвращает дайджест сообщения в виде байтового массива:

```
byte[] hash = alg.digest();
```

Программа, код которой приведен в листинге 9.15, вычисляет дайджест сообщения, используя алгоритм SHA или MD5. Данные можно загрузить из файла или ввести с помощью клавиатуры в текстовой области. На рис. 9.11 показано окно этой программы.

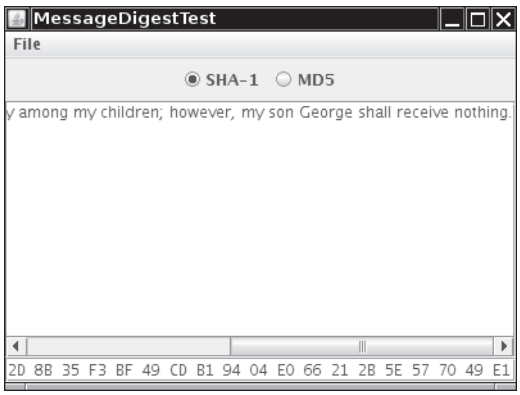


Рис. 9.11. Вычисление дайджеста сообщения

Листинг 9.15. Содержимое файла `MessageDigestTest.java`

```
1.  import java.io.*;
2.  import java.security.*;
3.  import java.awt.*;
4.  import java.awt.event.*;
5.  import javax.swing.*;
6.
7.  /**
8.   * Данная программа вычисляет дайджест сообщения
9.   * для файла или содержимого текстовой области.
10.  * @version 1.13 2007-10-06
11.  * @author Cay Horstmann
12.  */
13. public class MessageDigestTest
14. {
15.    public static void main(String[] args)
16.    {
17.      EventQueue.invokeLater(new Runnable()
18.       {
19.         public void run()
20.         {
21.           JFrame frame = new MessageDigestFrame();
22.           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.           frame.setVisible(true);
24.         }
25.       });
26.    }
27. }
```

```
28.  /**
29.   * Фрейм, содержащий меню, посредством которого запускается
30.   * процесс вычисления профиля сообщения для текстового файла
31.   * или содержимого текстовой области, переключатели для выбора
32.   * алгоритма SHA-1 или MD5, текстовую область, а также
33.   * поле редактирования для отображения профиля сообщения.
34.   */
35.  class MessageDigestFrame extends JFrame
36.  {
37.    public MessageDigestFrame()
38.    {
39.       setTitle("MessageDigestTest");
40.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
41.
42.       JPanel panel = new JPanel();
43.       ButtonGroup group = new ButtonGroup();
44.       addRadioButton(panel, "SHA-1", group);
45.       addRadioButton(panel, "MD5", group);
46.
47.       add(panel, BorderLayout.NORTH);
48.       add(new JScrollPane(message), BorderLayout.CENTER);
49.       add(digest, BorderLayout.SOUTH);
50.       digest.setFont(new Font("Monospaced", Font.PLAIN, 12));
51.
52.       setAlgorithm("SHA-1");
53.
54.       JMenuBar menuBar = new JMenuBar();
55.       JMenu menu = new JMenu("File");
56.       JMenuItem fileDigestItem = new JMenuItem("File digest");
57.       fileDigestItem.addActionListener(new ActionListener()
58.          {
59.             public void actionPerformed(ActionEvent event)
60.             {
61.                loadFile();
62.              }
63.          });
64.       menu.add(fileDigestItem);
65.       JMenuItem textDigestItem =
66.          new JMenuItem("Text area digest");
67.       textDigestItem.addActionListener(new ActionListener()
68.          {
69.             public void actionPerformed(ActionEvent event)
70.             {
71.                String m = message.getText();
72.                computeDigest(m.getBytes());
73.             }
74.          });
75.       menu.add(textDigestItem);
76.       menuBar.add(menu);
77.       setJMenuBar(menuBar);
78.    }
79.    /**
80.     * Добавление переключателей, предназначенных для
81.     * выбора алгоритма.
82.     * @param c Контейнер для размещения переключателей
83.     * @param name Имя алгоритма
84.     * @param g Группа переключателей
85.     */
```

```
86.    public void addRadioButton(Container c, final String name,
87.       ButtonGroup g)
88.    {
89.       ActionListener listener = new ActionListener()
90.          {
91.            public void actionPerformed(ActionEvent event)
92.            {
93.               setAlgorithm(name);
94.            }
95.          };
96.       JRadioButton b = new
97.          JRadioButton(name, g.getButtonCount() == 0);
98.       c.add(b);
99.       g.add(b);
100.      b.addActionListener(listener);
101.   }
102.
103.   /**
104.    * Установка алгоритма, используемого для вычисления дайджеста.
105.    * @param alg Имя алгоритма
106.    */
107.   public void setAlgorithm(String alg)
108.   {
109.      try
110.      {
111.         currentAlgorithm = MessageDigest.getInstance(alg);
112.         digest.setText("");
113.      }
114.      catch (NoSuchAlgorithmException e)
115.      {
116.         digest.setText("" + e);
117.      }
118.   }
119.
120.   /**
121.    * Загрузка файла и вычисление дайджеста сообщения.
122.    */
123.   public void loadFile()
124.   {
125.      JFileChooser chooser = new JFileChooser();
126.      chooser.setCurrentDirectory(new File("."));
127.
128.      int r = chooser.showOpenDialog(this);
129.      if (r == JFileChooser.APPROVE_OPTION)
130.      {
131.         try
132.         {
133.            String name = chooser.getSelectedFile().getAbsolutePath();
134.            computeDigest(loadBytes(name));
135.         }
136.         catch (IOException e)
137.         {
138.            JOptionPane.showMessageDialog(null, e);
139.         }
140.      }
141.   }
142.
```

```
143.    /**
144.     * Загрузка файла.
145.     * @param name Имя файла
146.     * @return Байтовый массив с данными из файла
147.     */
148.    public byte[] loadBytes(String name) throws IOException
149.    {
150.       FileInputStream in = null;
151.       in = new FileInputStream(name);
152.       try
153.       {
154.          ByteArrayOutputStream buffer = new ByteArrayOutputStream();
155.          int ch;
156.          while ((ch = in.read()) != -1)
157.             buffer.write(ch);
158.          return buffer.toByteArray();
159.       }
160.       finally
161.       {
162.          in.close();
163.       }
164.    }
165.
166.    /**
167.     * Вычисление дайджеста сообщения для байтового массива
168.     * и отображение его в поле редактирования.
169.     * @param b Байтовый массив, для которого вычисляется дайджест.
170.     */
171.    public void computeDigest(byte[] b)
172.    {
173.       currentAlgorithm.reset();
174.       currentAlgorithm.update(b);
175.       byte[] hash = currentAlgorithm.digest();
176.       String d = "";
177.       for (int i = 0; i < hash.length; i++)
178.       {
179.          int v = hash[i] & 0xFF;
180.          if (v < 16) d += "0";
181.          d += Integer.toString(v, 16).toUpperCase() + " ";
182.       }
183.       digest.setText(d);
184.    }
185.
186.    private JTextArea message = new JTextArea();
187.    private JTextField digest = new JTextField();
188.    private MessageDigest currentAlgorithm;
189.    private static final int DEFAULT_WIDTH = 400;
190.    private static final int DEFAULT_HEIGHT = 300;
191. }
```

API java.security.MessageDigest 1.1

- `static MessageDigest getInstance(String algorithm)`

  Возвращает объект `MessageDigest`, который реализует указанный алгоритм. Если такой алгоритм не поддерживается, выдает исключение `NoSuchAlgorithmExcetion`.

- `void update(byte input)`
- `void update(byte[] input)`
- `void update(byte[] input, int offset, int len)`

  Обновляет дайджест, используя указанные байты.

- `byte[] digest()`

  Выполняет вычисление по хеш-алгоритму, возвращает рассчитанный дайджест и затем переводит реализующий алгоритм объект в исходное состояние.

- `void reset()`

  Возвращает реализующий дайджест объект в исходное состояние.

Подписание сообщений

В предыдущем разделе было показано, как можно создавать дайджест сообщения, то есть делать своего рода дактилоскопический отпечаток исходного сообщения. В случае изменения сообщения отпечаток измененного сообщения не будет совпадать с отпечатком исходного. Это означает, что при доставке сообщения и еще отдельно его отпечатка получатель сможет проверять, не было ли оно подделано. Если, однако, злоумышленнику удастся перехватить как сообщение, так и его исходный отпечаток, он сможет легко изменить сообщение и переделать этот отпечаток так, как ему нужно. В конце концов, алгоритмы создания дайджестов сообщений всем известны и не требуют использования никаких секретных ключей. А в таком случае получатель поддельного сообщения и переделанного отпечатка, естественно, никогда не сможет узнать, что его сообщение было изменено. Избегать этой проблемы позволяют цифровые подписи.

Для понимания того, как работают цифровые подписи, сначала необходимо ознакомиться с несколькими концепциями из области *шифрования с открытым ключом* (public key cryptography). Основными в этой области являются такие понятия, как *открытый ключ* (public key) и *секретный ключ* (private key). Открытый ключ сообщается всем, а секретный держится в строгом секрете. Соответствие между этими ключами устанавливается на основании математических отношений, детали которых здесь не имеют значения. (Тем, кому интересно узнать о них более подробно, могут прочитать книгу *The Handbook of Applied Cryptography*, которая доступна по адресу `http://www.cacr.math.uwaterloo.ca/hac/`.)

Эти ключи являются довольно длинными и сложными. Например, ниже приведена пара соответствующих друг другу ключей, полученных с помощью алгоритма DSA (Digital Signature Algorithm — алгоритм создания цифровых подписей).

Открытый ключ:

```
p:
fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617ae01f35b91a47e6df
63413c5e12ed0899bcd132acd50d99151bdc43ee737592e17
q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5
g:678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d14271b9e35030b71fd73d
a179069b32e29356
30e
1c2062354d0da20a6c416e50be794ca4
y:
c0b6e67b4ac098eb1a32c5f8c4c1f0e7e6fb9d832532e27d0bdab9ca2d2a8123ce5a8018b8161a7
60480fadd040b927281ddb22cb9bc4df596d7de4d1b977d50
```

Соответствующий ему секретный ключ:

```
p:
fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617ae01f35b91a47e6df
63413c5e12ed0899bcd132acd50d99151bdc43ee737592e17
q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5
g:
678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d14271b9e35030b71fd73da1
79069b32e2935630e1c2062354d0da20a6c416e50be794ca4
x: 146c09f881656cc6c51f27ea6c3a91b85ed1d70a
```

Считается, что вывести один ключ из другого практически не возможно. То есть хотя открытый ключ известен всем, узнать с его помощью секретный ключ у злоумышленников никогда не получится, сколько бы ресурсов у них ни было в распоряжении.

Возможно, в то, что ни у кого не получится вычислить секретный ключ на основе открытого, и трудно поверить, но, по крайней мере, пока еще никому не удалось изобрести алгоритм, способный делать подобное с ключами, генерируемыми с помощью наиболее популярных сегодня механизмов шифрования. Из-за того, что эти ключи являются достаточно длинными, для их расшифровки путем применения метода грубой силы — то есть просто путем перебора всех возможных значений — потребуется компьютеров больше, чем может быть вообще создано из всех атомов солнечной системы, и не одна тысяча лет. Конечно, не исключено, что кому-то удастся создать для вычисления ключей более эффективный, чем метод грубой силы, алгоритм. Например, возьмем алгоритм RSA, который был изобретен Райвестом (Rivest), Шамиром (Shamir) и Адлеманом (Adleman). Действие этого алгоритма основывается на сложности разложения больших чисел на простые множители. За последние 20 лет многие известные математики пытались разработать хорошие алгоритмы разложения чисел на простые множители, но пока что никому это не удалось. Из-за этого большинство специалистов по шифрованию считают, что в настоящее время ключи с размером 2000 битов являются абсолютно защищенными от любых попыток взлома. Алгоритм DSA тоже считается довольно надежным.

На рис. 9.12 показано, как процесс работы DSA выглядит на практике.

Предположим, что Алиса хочет послать сообщение Бобу, а Боб желает быть уверенным в том, сообщение поступило именно от Алисы, а не от какого-то самозванца. Для этого Алиса пишет сообщение и затем *подписывает* (sign) дайджест этого сообщения с помощью своего секретного ключа. Далее Боб получает копию ее открытого ключа и применяет его для *верификации* (verify) подписи. В случае успешного прохождения процесса верификации, Боб может быть уверен в двух следующих вещах:

- что исходное сообщение не было изменено;
- что сообщение было подписано Алисой, владелицей секретного ключа, соответствующего тому открытому ключу, которое Боб использовал для верификации.

Это позволяет увидеть, почему секретные ключи являются настолько важными. Если кто-то выкрадет секретный ключ Алисы или если правительство попросит ее расшифровать его, ей грозят неприятности, потому что выкравший ключ злоумышленник или представитель правительства тогда сможет легко подделывать ее сообщения, заявки не пересылку денежных средств и выполнять другие подобные операции, а все будут думать, что это делает сама Алиса.

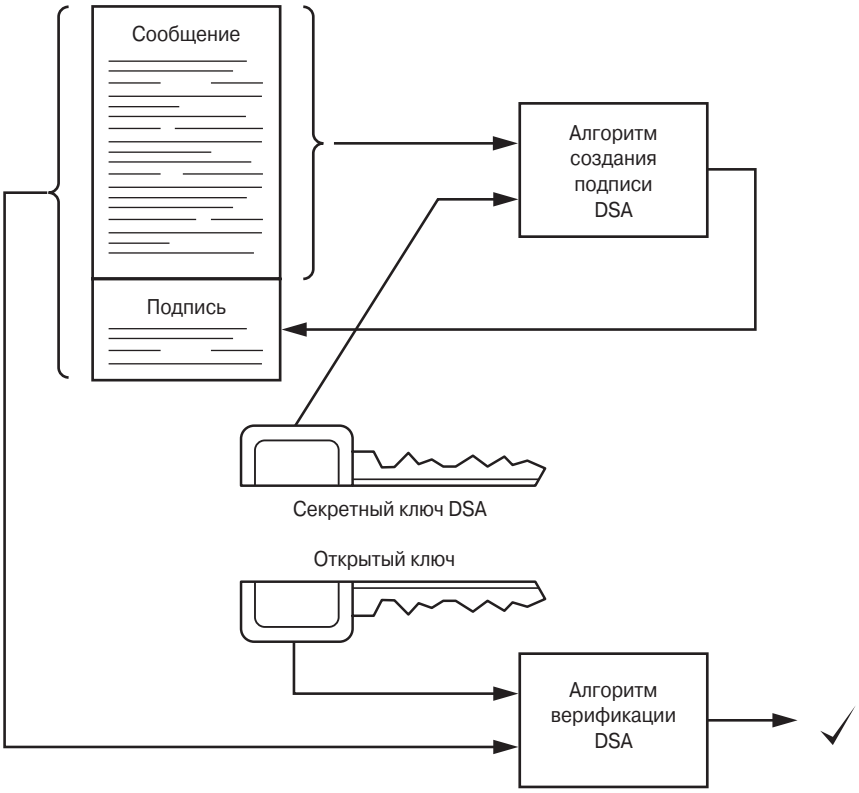


Рис. 9.12. Обмен сообщениями с использованием цифровой подписи, открытых ключей и алгоритма DSA

Формат сертификата X.509

Для извлечения пользы из технологии шифрования с помощью открытого ключа открытые ключи необходимо как-то распространять. Одним из наиболее популярных форматов для распространения является X.509. Сертификаты в формате X.509 широко применяются Verisign, Microsoft, Netscape и другими компаниями для подписания электронных сообщений, аутентификации программного кода и сертификации многих других видов данных. Стандарт X.509 является частью набора рекомендаций X.500, разработанных Консультативным комитетом по международной телефонной и телеграфной связи США (Consultative Committee for International Telephone and Telegraphy – CCITT).

Точное описание структуры сертификата X.509 содержится в официальном документе ASN.1 (Abstract Syntax Notation #1). На рис. 9.13 показано, как выглядит определение ASN.1 для третьей версии формата X.509. Детали синтаксиса нам не важны, но зато описание структуры файла сертификата в ASN.1 выглядит очень точно. *Базовые правила шифрования* (basic encoding rules – BER) и являющиеся их разновидностью *отличительные правила шифрования* (distinguished encoding rules – DER) четко описывают, как сохранять эту структуру в двоичном файле. То есть они описывают, как необходимо шифровать целые числа, символьные строки, битовые строки и конструкции, вроде `SEQUENCE`, `CHOICE` и `OPTIONAL`.

```
[Certificate ::= SEQUENCE {
      tbsCertificate           TBSCertificate,
      signatureAlgorithm       AlgorithmIdentifier,
      signature                BIT STRING }

  TBSCertificate ::= SEQUENCE {
      version         [0]      EXPLICIT Version DEFAULT v1,
      serialNumber             CertificateSerialNumber,
      signature                AlgorithmIdentifier,
      issuer                   Name,
      validity                 Validity,
      subject                  Name,
      subjectPublicKeyInfo     SubjectPublicKeyInfo,
      issuerUniqueID  [1]      IMPLICIT UniqueIdentifier OPTIONAL,
                               -- If present, version must be v2 or v3
      subjectUniqueID [2]      IMPLICIT UniqueIdentifier OPTIONAL,
                               -- If present, version must be v2 or v3
      extensions      [3]      EXPLICIT Extensions OPTIONAL
                               -- If present, version must be v3
      }

  Version ::= INTEGER { v1(0), v2(1), v3(2) }

  CertificateSerialNumber ::= INTEGER

  Validity ::= SEQUENCE {
      notBefore       CertificateValidityDate,
      notAfter        CertificateValidityDate }

  CertificateValidityDate ::= CHOICE {
      utcTime         UTCTime,
      generalTime     GeneralizedTime }

  UniqueIdentifier ::= BIT STRING

  SubjectPublicKeyInfo ::= SEQUENCE {
      algorithm               AlgorithmIdentifier,
      subjectPublicKey        BIT STRING }

  Extensions ::= SEQUENCE OF Extension

  Extension ::= SEQUENCE {
      extnID      OBJECT IDENTIFIER,
      critical    BOOLEAN DEFAULT FALSE,
      extnValue   OCTET STRING }
```

Рис. 9.13. Определение X.509v3 в ASN.1



Дополнительную информацию об ASN.1 можно найти в книге Бартона С. Калиски (Burton S. Kaliski) *A Layman's Guide to a Subset of ASN.1, BER, and DER* (которая доступна по адресу `ftp://ftp.rsa.com/pub/pkcs/ps/layman.ps`), в книге *ASN.1-Communication Between Heterogeneous Systems* Оливера Дюбиссона (Olivier Dubuisson) (Academic Press, 2000 г.) (`http://www.oss.com/asn1/dubuisson.html`), а также в работе Джона Лармоута (John Larmouth) *ASN.1 Complete* (Morgan Kaufmann Publishers, 1999 г.) (`http://www.nokalva.com/asn1/larmouth.html`).

Проверка подписи

В состав JDK входит программа **keytool**, которая представляет собой утилиту командной строки и предназначена для генерации и управления сертификатами. Пока она может работать только в режиме командной строки, но мы надеемся, что в следующих выпусках JDK ее функциональные возможности будут доступны и в виде других, более дружественных к пользователю версиях программ. В этом разделе мы используем ее для того, чтобы показать, как Алиса может подписывать документ и отправлять его Бобу, а Боб — проверять и удостоверяться в том, что документ был действительно подписан Алисой, а не каким-то самозванцем.

Программа `keytool` позволяет управлять *хранилищами ключей* (key store), базами данных сертификатов и парами секретных и открытых ключей. У каждой записи в хранилище ключей имеется свой *псевдоним* (alias). Ниже показано, как Алиса может создать хранилище `alice.store` и сгенерировать пару ключей с псевдонимом `alice`:

```
keytool - genkeypair -keystore alice.certs -alias alice
```

При создании или открытии хранилища появляется приглашение ввести пароль. В данном примере предлагается ввести просто слово `secret`. В случае создания хранилища ключей с помощью `keytool` для каких-то более серьезных целей, конечно же, нужно выбрать надежный пароль и хранить его в полном секрете от всех. При генерировании ключа появляется приглашение ввести следующую информацию:

```
Enter keystore password: secret
Reenter new password: secret
What is your first and last name?
   [Unknown]: Alice Lee
What is the name of your organizational unit?
   [Unknown]: Engineering Department
Which is the name of your organization?
   [Unknown]: ACME Software
Which is the name of your City or Locality?
   [Unknown]: San Francisco
What is the name of your State or Province?
   [Unknown]: CA
What is the two-letter country code for this unit?
   [Unknown]: US
Is <CN=Alice Lee, OU=Engineering Department, O=ACME Software, L=San Francisco,
ST=CA, C=US> correct?
   [no]: yes
```

В программе **keytool** для идентификации владельцев ключей и создателей сертификатов используются имена стандарта X.500 с компонентами 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, наконец, напоследок требуется указать пароль для ключа или нажать клавишу <Enter>, чтобы использовать в качестве оного пароль хранилища ключей.

Предположим, что Алиса желает предоставить копию своего открытого ключа Бобу. В таком случае ей необходимо сначала экспортировать файл сертификата:

```
keytool -export -keystore alice.store -alias alice -file alice.cert
```

После этого она может отправить этот сертификат Бобу. Боб же после получения этого сертификата может распечатать его с помощью следующей команды:

```
keytool -printcert -file alice.cer
```

Выходные данные после выполнения этой команды будут выглядеть следующим образом:

```
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
```

Если Боб захочет проверить, тот ли сертификат он получил, он может позвонить Алисе и сверить свою копию с исходным отпечатком по телефону.



Некоторые создатели сертификатов выкладывают отпечатки сертификатов на Web-сайтах. Например, чтобы проверить сертификат VeriSign из каталога хранилища ключей `jre/lib/security/cacerts`, можно сначала отобразить содержимое этого каталога с помощью опции `-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
```

Дабы удостовериться в том, что данный сертификат является действительным, достаточно будет просто посетить Web-сайт `http://www.verisign.com/repository/root.html`.

Убедившись в правильности сертификата, Боб далее может импортировать его в свое хранилище ключей:

```
keytool -importcert -keystore bob.certs -alias alice -file alice.cer
```



Никогда не следует импортировать в хранилище ключей непроверенный сертификат. После добавления сертификата в хранилище любая пользующаяся хранилищем ключей программа предполагает, что данный сертификат может применяться для проверки подписей.

После этого Алиса может начинать посылать Бобу подписанные документы. Подписывать и проверять JAR-файлы позволяет утилита `jarsigner`. Сначала Алисе нужно просто добавить подлежащий подписанию документ в файл JAR:

```
jar cvf document.jar document.txt
```

Далее с помощью программы `jarsigner` она может добавить к этому файлу свою подпись. Для этого ей потребуется указать хранилище ключей, JAR-файл и псевдоним подлежащего использованию ключа:

```
jarsigner -keystore alice.certs document.jar alice
```

При получении этого файла Боб может использовать для его верификации параметр `-verify`:

```
jarsigner -verify -keystore bob.certs document.jar
```

Указывать псевдоним ключа при этом Бобу не нужно. Программа `jarsigner` самостоятельно отыщет указанное в формате X.509 имя владельца ключа в цифровой подписи и выполнит поиск соответствующих ему сертификатов в хранилище ключей.

Если JAR-файл не поврежден и подпись совпадает, тогда после этого программа `jarsigner` отобразит следующее сообщение:

```
jar verified.
```

В противном случае вместо этого сообщения появится сообщение об ошибке.

Проблема аутентификации

Предположим, что вы получаете сообщение от своей подруги Алисы, которая подписала его описанным выше методом с помощью своего секретного ключа. У вас уже может иметься копия ее открытого ключа, а если нет, то вы запросто можете раздобыть ее либо у самой Алисы, либо на ее Web-странице. Наличие этого ключа позволяет легко удостовериться в том, что его 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 воспользоваться прикрепленной им к данному сообщению программой. Этот незнакомец даже присылает вам копию своего открытого ключа, дабы вы могли удостовериться в том, что именно он является автором данного сообщения. Вы проверяете и видите, что подпись на самом деле является действительной. Это подтверждает лишь то, что сообщение было подписано с помощью соответствующего секретного ключа и что оно не было подделано.

Будьте осторожны: *ведь 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. Проблема выяснения личности отправителя называется *задачей аутентификации* (authentication problem).

Обычно проблема аутентификации решается 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 этот 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 (рис. 9.14). В таком случае получается, что гарантом аутентичности отправителя будет ваш знакомый.

На самом деле вашему знакомому даже вовсе и 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. 9.15).

При получении файла с открытым ключом вы можете проверить подпись своего друга и, поскольку вы ему доверяете, быть уверенным в том, что прежде чем добавлять свою подпись, он обязательно проверил учетные данные незнакомца.

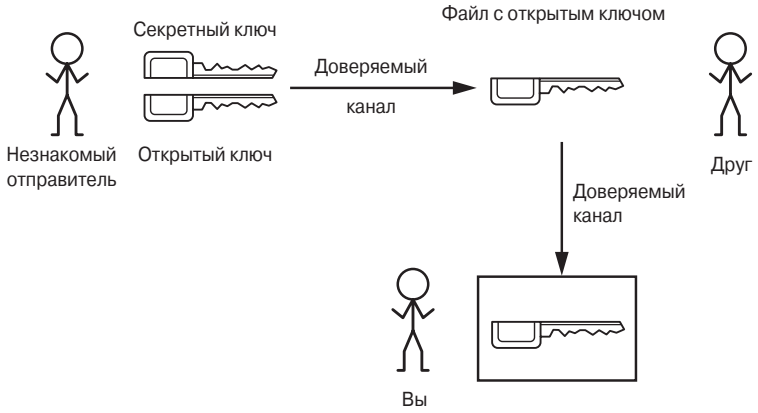


Рис. 9.14. Аутентификация через доверенного посредника

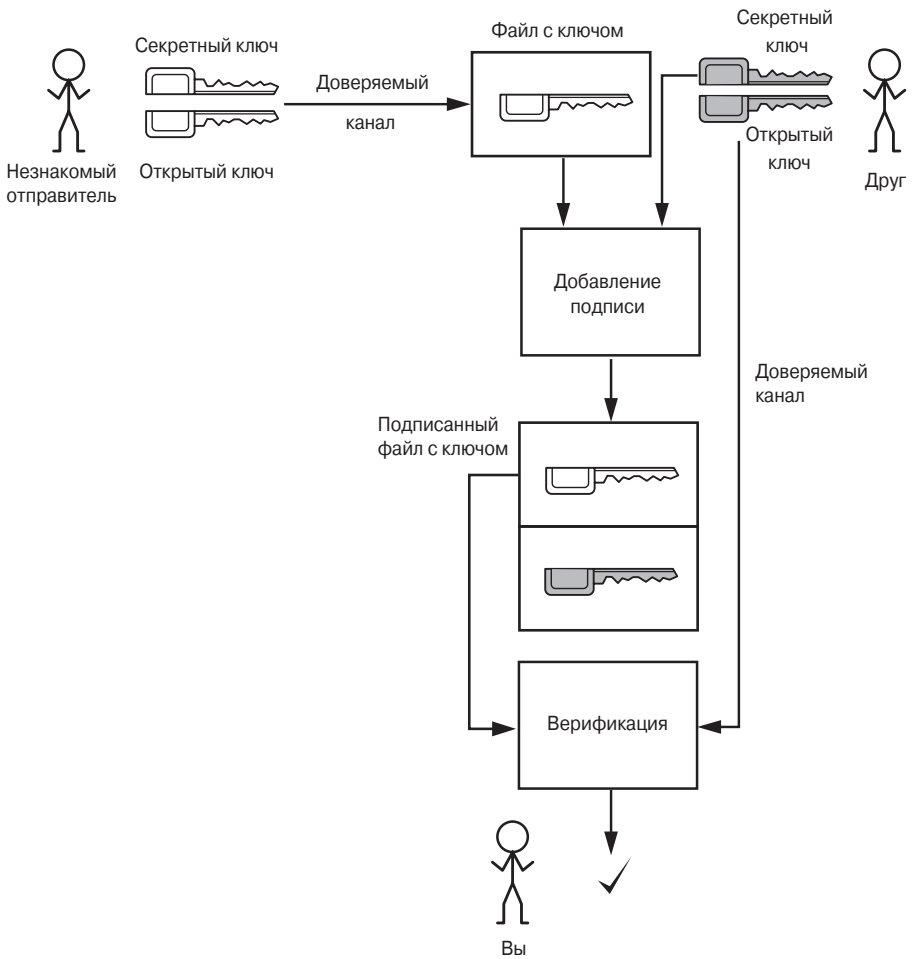


Рис. 9.15. Аутентификация через подпись доверенного посредника

Однако у вас может и не быть такого общего знакомого. В некоторых моделях доверительных отношений предполагается, что всегда существует некая "цепочка доверия" ("chain of trust"), т.е. цепочка общих знакомых, всем членам которой можно спокойно доверять. На практике, конечно, так бывает не всегда. Вы можете доверять своей подруге Алисе, и знать, что Алиса доверяет Бобу, но сами-то вы Боба не знаете и, следовательно, не можете быть уверенными в том, стоит ему доверять или нет. Другая модель доверительных отношений предполагает наличие какого-то одного хорошо зарекомендовавшего себя человека или компанию, которой все могут безоговорочно доверять. Наиболее известным примером такой модели является компания Verisign, Inc. (`www.verisign.com`).

Вам часто будут встречаться цифровые подписи, заверенные одной или несколькими ручающимися за аутентичность организациями, и будет нужно самостоятельно оценивать то, насколько им стоит доверять. Например, вы можете доверять VeriSign, к примеру, из-за того, что часто встречаете ее логотип на многих Web-страницах или из-за того, что слышали, будто бы в ней применяются невероятные меры безопасности.

Однако вы все равно должны иметь реальное представление о том, как именно в таких компаниях проводится процесс аутентификации и что именно они проверяют. Например, что касается компании VeriSign, то естественно, что ее исполнительный директор лично с каждым человеком или представителем компании при аутентификации ключа не встречается. Получить идентификатор "класса 1" от этой компании можно, просто заполнив Web-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 предоставления отчета о финансовом положении, официально заверенного у нотариуса. У других компаний будут другие требования. Поэтому очень важно, чтобы при получении заверенного сообщения вы понимали, на предмет чего именно оно проверялось.

Подписание сертификатов

В разделе "Проверка подписи" было показано, как Алиса использовала самостоятельно подписываемый сертификат для передачи открытого ключа Бобу. Однако Бобу при этом требовалось проверять полученный сертификат на предмет действительности путем выполнения его сверки с исходным отпечатком Алисы.

Теперь представим, что Алиса хочет послать подписанное сообщение своей коллеге Синди, а та не любит тратить время на выполнение сверки подписей с исходными отпечатками. В таком случае может помочь участие некой организации, которой бы Синди могла доверить проверку всех подписей. В рассматриваемом здесь примере предполагается, что такой организацией является отдел информационных ресурсов Informational Resources Department компании ACME Software.

Этот отдел занимается выдачей сертификатов, то есть представляет собой своего рода *центр сертификации* (Certificate Authority — CA). У каждого сотрудника в ACME имеется полученный от этого отдела открытый ключ, который находится в хранилище ключей, и перед установкой был тщательно сверен с отпечатком системным администратором. Этот CA еще и подписывает ключи всех сотрудников ACME. Благодаря этому, при установке ключей друг друга хранилище ключей автоматически доверяет им, поскольку они подписаны с помощью доверяемого ключа.

Ниже показано, как можно сымитировать этот процесс. Создайте хранилище ключей `acmesoft.certs`. Далее сгенерируйте пару ключей и экспортируйте открытый ключ:

```
keytool -genkeypair -keystore acmesoft.certs -alias acmeroot
keytool -exportcert -keystore acmesoft.certs -alias acmeroot -file acmeroot.cer
```

Открытый ключ будет экспортирован в "самостоятельно подписываемый сертификат". Далее добавьте его в хранилище ключей каждого сотрудника:

```
keytool -importcert -keystore cindy.certs -alias acmeroot -file acmeroot.cer
```

Чтобы получить возможность отправлять подписанные сообщения Синди и любым другим сотрудникам ACME Software, Алисе необходимо принести свой сертификат в отдел информационных ресурсов компании ACME Software и подписать его. Программа `keytool`, к сожалению, не предоставляет никаких средств для решения подобной задачи. В загружаемом коде для устранения упомянутого пробела предлагается класс `CertificateSigner`. Уполномоченному сотруднику компании ACME Software далее остается только проверить личность Алисы и сгенерировать подписанный сертификат с помощью следующей команды:

```
java CertificateSigner -keystore acmesoft.certs -alias acmeroot
  -infile alice.cer -outfile alice_signedby_acmeroot.cer
```

Очевидно, что выполнение этой команды является довольно щепетильной операцией 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 желаемому пользователю в компании ACME Software. В качестве альтернативного варианта, этот файл еще также может быть и просто сохранен в каталоге компании. Напомним, что в этом файле содержится открытый ключ Алисы и подтверждение компании ACME Software о том, что данный ключ действительно принадлежит Алисе. Далее Синди может импортировать подписанный сертификат Алисы в свое хранилище ключей с помощью такой команды:

```
keytool -importcert -keystore cindy.certs -alias alice
  -file alice_signedby_acmeroot.cer
```

Запуск этой команды приведет к автоматическому выполнению хранилищем ключей проверки на предмет того, был ли данный ключ подписан с помощью уже представленного в нем доверяемого корневого ключа. Благодаря этому Синди не потребуется выполнять его сверку с исходным отпечатком этого сертификата.

Добавив корневой сертификат и сертификаты всех остальных людей, которые часто присылают документы один раз, Синди больше никогда не придется снова волноваться о хранилище ключей.

Запросы сертификатов

В предыдущем разделе был показан центр сертификации (CA) с хранилищем ключей и программой `CertificateSigner`. Однако в большинстве CA для управления сертификатами применяется более сложное программное обеспечение и немного другие форматы сертификатов. В этом разделе демонстрируются дополнительные шаги, которые требуются для взаимодействия с такими пакетами программного обеспечения.

Мы в качестве примера будем использовать пакет программного обеспечения OpenSSL. Это программное обеспечение обычно заранее устанавливается для многих систем Linux и Mac OS X вместе с портом Cygwin. Если оно не установлено, его всегда можно легко загрузить с Web-сайта по адресу `http://www.openssl.org`.

Чтобы создать CA, нужно запустить сценарий CA. Точное место размещения этого сценария зависит от того, о какой именно операционной системе идет речь. В системе Ubuntu, например, его можно запустить с помощью следующей команды:

```
/usr/lib/ssl/misc/CA.pl -newca
```

Этот сценарий создаст в текущем каталоге подкаталог с именем `demoCA`, содержащий пару корневых ключей и хранилище для сертификатов и списков аннулирования сертификатов.

Далее потребуется импортировать открытый ключ в Java-хранилище ключей всех сотрудников, но из-за того, что он имеет формат PEM (Privacy Enhanced Mail), 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 идет перед строкой `-----BEGIN CERTIFICATE-----` и после строки `-----END CERTIFICATE-----`.

После этого `acmeroot.pem` можно скопировать в каждое хранилище ключей обычным образом:

```
keytool -importcert -keystore cindy.certs -alias alice -file acmeroot.pem
```

Кажется невероятным, что утилита `keytool` не способна выполнить эту операцию по редактированию самостоятельно.

Чтобы подписать открытый ключ Алисы, сначала следует сгенерировать запрос сертификата, содержащий сертификат в формате PEM:

```
keytool -certreq -keystore alice.store -alias alice -file alice.pem
```

Далее, чтобы подписать этот сертификат, нужно выполнить следующую команду:

```
openssl ca -in alice.pem -out alice_signedby_acmeroot.pem
```

Как и прежде, потом потребуется удалить из файла `alice_signedby_acmeroot.pem` все строки, которые находятся за пределами маркеров `BEGIN CERTIFICATE/END CERTIFICATE` и затем импортировать его в хранилище ключей:

```
keytool -importcert -keystore cindy.certs -alias alice
  -file alice_signedby_acmeroot.pem
```

Все эти же шаги нужно будет выполнять и в случае подписания сертификата у какой-то внешней компании вроде VeriSign.

Подписание кода

Одно из самых важных применений технологии аутентификации состоит в подписании выполняемых файлов. 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 необходимо быть уверенным в том, что программа выслана надежным источником *и* во время пересылки не была изменена. Если программа написана на языке Java, то, обладая такой информацией, можно принять решение о делегировании ей соответствующих полномочий. Программу можно запустить в “песочнице” как обычный аплет, либо предоставить ей дополнительные права и наложить ограничения. Например, если скопированная из сети программа предназначена для редактирования текста, то ей можно разрешить обращаться к принтеру и к файлам из определенного каталога. Этой программе можно запретить доступ к сетевым соединениям, чтобы она не могла без разрешения рассылать ваши файлы третьим лицам.

Для реализации такой сложной схемы нужно выполнить перечисленные ниже действия.

1. С помощью средств аутентификации проверить источник, из которого получен код.
2. Запустить программу с соответствующей политикой защиты, которая определяет права программы.

Подписание JAR-файлов

В данном разделе показано, как можно подписывать аплеты и приложения Web Start для использования с программным обеспечением Java Plug-In, на примере двух следующих сценариев:

- с доставкой по внутренней сети;
- с доставкой через Internet.

В первом сценарии системный администратор устанавливает файлы политик и сертификаты на локальных компьютерах. После этого программа Java Plug-in при каждой загрузке подписанного кода обращается к файлу политики для проверки полномочий, а к хранилищу ключей — для проверки подписей. Процесс установки сертификатов и политик выглядит довольно просто и может выполняться всего лишь один раз для каждой настольной системы. После его выполнения конечные пользователи получают возможность запускать подписанный корпоративный код за пределами песочницы. При создании каждой новой программы или обновлении какой-нибудь существующей, она должна обязательно подписываться и развертываться на Web-сервере. Однако изменять настольные системы при таком совершенствовании программ совершенно не требуется. На наш взгляд, такой сценарий является вполне целесообразным и может быть довольно привлекательной альтернативной отдельному развертыванию корпоративных приложений на каждой настольной системе.

Во втором сценарии поставщики программного обеспечения получают сертификаты, подписанные компаниями, вроде Verisign. При посещении пользователем Web-сайта, содержащего подписанный аплет, на экране появляется диалоговое окно, указывающее на поставщика данного программного обеспечения и предлагающее пользователю на выбор выполнить одно из следующих действий: либо запустить аплет с полным набором привилегий, либо ограничить его область действия песочницей. Более подробно этот менее желательный сценарий будет рассматриваться чуть позже в этой главе, в разделе “Сертификаты разработчиков программ”.

В остальной части этого раздела объясняется, как можно создавать файлы политики, предоставляющие коду из известных источников только конкретные полномочия. Процесс создания и развертывания таких файлов политик не может выполняться обычными конечными пользователями. Системным администраторам, однако, разрешается его осуществлять в качестве подготовки для распространения программ по внутренней корпоративной сети.

Давайте предположим, что компания ACME Software желает, чтобы ее пользователи могли запускать определенные программы, требующие доступа к локальным файлам, и хочет сделать так, чтобы эти программы были доступны через браузер в виде аплетов или приложений Web Start. Поскольку работать в пределах песочницы такие программы не могут, компания ACME Software вынуждена установить файлы политики на компьютерах сотрудников.

Как уже показывалось ранее в этой главе, компания ACME могла бы идентифицировать программы по их кодовой базе. Однако это означает, что ей тогда нужно было бы обновлять файлы политик при каждом перемещении программ на другой сервер. Поэтому вместо такого подхода ACME принимает решение *подписывать* JAR-файлы, содержащие программный код.

Для этого ACME сначала генерирует корневой сертификат:

```
keytool -genkeypair -keystore acmesoft.certs -alias acmeroo
```

Разумеется, хранилище ключей, содержащее секретный корневой ключ, должно обязательно находиться в безопасном месте. Поэтому мы создаем второе хранилище ключей `client.certs` для размещения в нем открытых сертификатов и сразу же добавляем в него открытый сертификат `acmeroot`:

```
keytool -exportcert -keystore acmesoft.certs -alias acmeroot -file acmeroot.cer
keytool -importcert -keystore client.certs -alias acmeroot -file acmeroot.cer
```

Для создания подписанного JAR-файла программисты могут добавлять свои файлы классов в JAR-файл обычным образом. Например:

```
javac FileReadApplet.java
jar cvf FileReadApplet.jar *.class
```

Далее уполномоченный сотрудник в ACME может запускать программу `jarsigner` и указывать JAR-файл и псевдоним секретного ключа:

```
jarsigner -keystore acmesoft.certs FileReadApplet.jar acmeroot
```

После этого аплет готов к развертыванию на Web-сервере.

Теперь давайте обратим внимание на конфигурацию на стороне клиента. Файл политики должен рассылаться каждой клиентской машине.

Для указания на хранилище ключей, файл политики начинается с такой строки:

```
keystore "URL-адрес_хранилища_ключей", "тип_хранилища_ключей";
```

URL-адрес может быть как абсолютным, так и относительным. Относительные URL-адреса являются относительными в отношении места размещения файла политики. Если хранилище генерировалось с помощью программы `keytool`, его тип будет выглядеть как `JKS`. Например:

```
keystore "client.certs", "JKS";
```

Далее, конструкции `grant` в файле политик могу сопровождаться суффиксами `signedBy "`*`псевдоним`*`"`, наподобие того, как показано ниже:

```
grant signedBy "acmeroot"
{
  ...
};
```

Внутри этих конструкций предоставляются права любому коду, который может быть проверен с помощью ассоциируемого с указанным псевдонимом открытого ключа.

Вы можете испробовать процесс подписания кода на примере аплета, приведенного в листинге 9.16. Этот аплет пытается прочитать локальный файл. Политика безопасности по умолчанию позволяет ему считывать файлы только из каталога кодовой базы и любых его подкаталогов. Воспользуйтесь программой просмотра аплетов `appletviewer`, чтобы запустить этот аплет и удостовериться в том, что с его помощью можно просматривать файлы только из каталога и подкаталогов кодовой базы и больше никаких других каталогов.

Теперь создайте файл политики `applets.policy` со следующим содержимым:

```
keystore "client.certs", "JKS";
grant signedBy "acmeroot"
{
  permission java.lang.RuntimePermission "usePolicy";
  permission java.io.FilePermission "/etc/*", "read";
};
```

Здесь полномочие `usePolicy` переопределяет используемое по умолчанию для подписанных аплетов полномочие типа "все или ничего" и указывает, что всем аплетам, которые подписаны с помощью `acmeroot`, разрешено считывать файлы из каталога `/etc` (в случае пользователей Windows этот каталог нужно заменить на какой-нибудь другой, например, на `C:\Windows`).

Теперь осталось только указать программе просмотра аплетов, чтобы она использовала этот файл политики:

```
appletviewer -J-Djava.security.policy=applets.policy
  FileReadApplet.html
```

После этого аплет сможет считывать файлы из каталога `/etc` и тем самым демонстрировать, как работает механизм подписания.

В качестве заключительной проверки вы можете попробовать запустить аплет в среде браузера (рис. 9.16). Для этого вам потребуется скопировать файл полномочий и хранилище ключей в каталог развертывания Java. В UNIX и Linux этот каталог содержится внутри рабочего каталога и выглядит как `.java/deployment`, а в Windows Vista — как `C:\Users\`*`регистрационное_имя_пользователя`*`\AppData\Sun\Java\Deployment`. В дальнейшем для удобства будем называть его общим именем *`deploydir`*.

Скопируйте файлы `applet.policy` и `client.certs` в каталог `deploydir/security`. В этом каталоге переименуйте файл `applets.policy` в `java.policy`. (Дважды перепроверьте, что это не приведет к удалению существующего файла `java.policy`. Если таковой имеется, тогда просто добавьте содержимое файла `applet.policy` в него.)

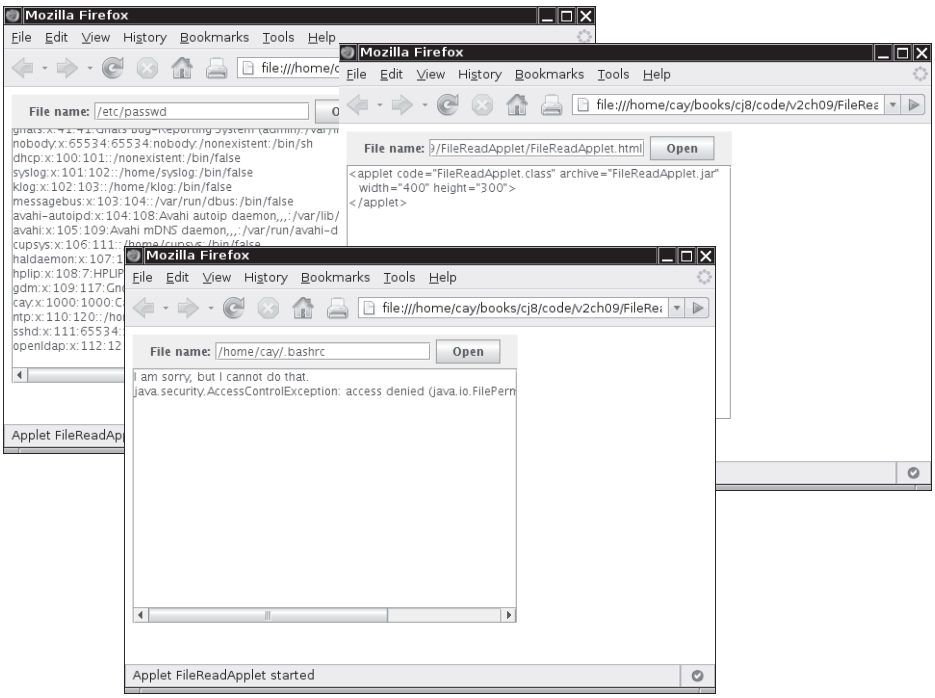


Рис. 9.16. Подписанный аплет имеет доступ к локальным файлам



Более подробную информацию о конфигурировании клиентских параметров безопасности Java можно найти в руководстве по развертыванию Java в разделах *Deployment Configuration File and Properties* и *Java Control Panel* (`http://java.sun.com/javase/6/docs/technotes/guides/deployment/deployment-guide/overview.html`.).

Перезапустите браузер и загрузите `FileReadApplet.html`. Никакое приглашение на принятие сертификата после этого появляться *не* должно. Далее удостоверьтесь в том, что из каталога `/etc` и каталога аплета *можно* загружать файлы, а из всех остальных каталогов — нет.

По завершении не забудьте очистить каталог `deploydir/security`. Удалите файлы `java.policy` и `client.certs`. Перезапустите браузер. Если после очистки вы снова попробуете загрузить аплет, возможности считывать файлы из локальной файловой системы уже быть не должно. Вместо этого на экране появится приглашение на принятие сертификата. О сертификатах более подробно будет рассказываться уже в следующем разделе.

Листинг 9.16. Содержимое файла `FileReadApplet.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.io.*;
4.  import java.util.*;
5.  import javax.swing.*;
6.
7.  /**
8.   * Аплет, который может работать за пределами "песочницы"
9.   * и при наличии соответствующих прав доступа обращаться
10.  * к локальным файлам.
11.  * @version 1.11 2007-10-06
12.  * @author Cay Horstmann
13.  */
14. public class FileReadApplet extends JApplet
15. {
16.   public void init()
17.   {
18.     EventQueue.invokeLater(new Runnable()
19.      {
20.       public void run()
21.       {
22.         fileNameField = new JTextField(20);
23.         JPanel panel = new JPanel();
24.         panel.add(new JLabel("File name:"));
25.         panel.add(fileNameField);
26.         JButton openButton = new JButton("Open");
27.         panel.add(openButton);
28.         openButton.addActionListener(new ActionListener()
29.          {
30.            public void actionPerformed(ActionEvent event)
31.            {
32.               loadFile(fileNameField.getText());
33.            }
34.          });
35.         fileNameField.addActionListener(listener);
36.         openButton.addActionListener(listener);
37.         add(panel, "North");
```

```
38.         fileText = new JTextArea();
39.         add(new JScrollPane(fileText), "Center");
40.       }
41.     });
42.  }
43.  /**
44.   * Загрузка содержимого файла в текстовую область.
45.   * @param filename Имя файла
46.   */
47.    public void loadFile(String filename)
48.    {
49.      try
50.      {
51.        fileText.setText("");
52.        Scanner in = new Scanner(new FileReader(filename));
53.        while (in.hasNextLine())
54.          fileText.append(in.nextLine() + "\n");
55.        in.close();
56.      }
57.      catch (IOException e)
58.      {
59.        fileText.append(e + "\n");
60.      }
61.      catch (SecurityException e)
62.      {
63.        fileText.append("I am sorry, but I cannot do that.\n");
64.        fileText.append(e + "\n");
65.      }
66.    }
67.
68.    private JTextField fileNameField;
69.    private JTextArea fileText;
70.  }
```

Сертификаты разработчиков программ

До сих пор обсуждались сценарии рассылки программ по корпоративной сети. В этом случае системный администратор настраивает политику защиты, управляющую привилегиями программ. Такая стратегия работает только с программами из известных источников.

Предположим, что, работая в глобальной сети, пользователь открывает Web-страницу, в которой предлагается запустить аплет или приложение Web Start неизвестного поставщика, как, например, показано на рис. 9.17. Такой аплет подписан сертификатом *разработчика программного обеспечения*. В диалоговом окне указывается разработчик данной программы и создатель сертификата. Таким образом, у пользователя есть два варианта продолжения работы.

- Запустить аплет с полными правами доступа.
- Ограничить работу аплета песочницей. (Название кнопки Cancel (Отмена) в диалоговом окне вводит в заблуждение. Щелчок на ней не приводит к отмене запуска аплета. Вместо этого он обеспечивает его запуск в пределах песочницы.)

Попробуем выяснить, какой информацией обладает пользователь в такой ситуации и что повлияет на его решение?

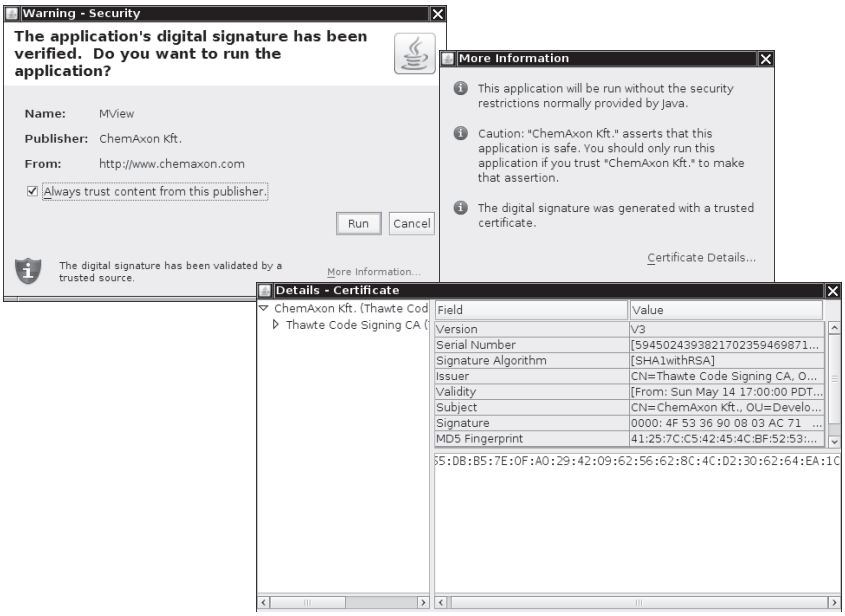


Рис. 9.17. Запуск подписанного аплета

- 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 он был проверен с использованием 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 какой-то злоумышленник. Более того, создатели сертификатов не претендуют на это, т.е. они не занимаются проверкой честности и компетентности поставщиков компьютерного обеспечения.

В ситуации, когда имя поставщика данного программного обеспечения неизвестно пользователю, нужно тщательно взвесить все аргументы за и против, прежде чем предоставлять данной программе права доступа. А если имя поставщика данного программного обеспечения хорошо известно пользователю, то он может принять решение, учитывая репутацию компании.



Некоторые сертификаты на проверку оказываются совершенно неубедительными. Обратившись по адресу `http://www.dallaway.com/acad/webstart` и ознакомившись с приведенным там примером, вы убедитесь, что не каждый сертификат является гарантией качества поступающей к вам информации. Некоторые разработчики даже рекомендуют включать сомнительные сертификаты в хранилище; подробно об этом можно прочитать в документе `http://www.agsrhichome.bnl.gov/Controls/doc/javaws/javaws_howto.html`. Однако с точки зрения безопасности такие действия совершенно недопустимы.

Вряд ли стоит доверять программе, которая заявляет "дай мне все права, или я вообще не будут 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 общедоступной сети Internet старались функционировать в пределах своих песочниц, и если бы эти песочницы совершенствовались. Первым шагом в верном направлении является API-интерфейс для приложений Web Start, о котором рассказывалось в главе 10 первого тома этой книги.

Шифрование

До сих пор рассматривалась аутентификация с помощью цифровых подписей. Еще одним важным средством обеспечения безопасности является *шифрование* (*encryption*). Информация, отмеченная цифровой подписью, доступна для просмотра, а подпись лишь подтверждает, что эта информация не была изменена. Для просмотра зашифрованных данных этого недостаточно и их нужно расшифровать с помощью соответствующего ключа.

Часто бывает так, что аплет или приложение содержат конфиденциальную информацию, например, номера кредитных карточек или другие личные данные, которые необходимо скрыть от посторонних. Именно в таких случаях используются технологии шифрования.

До недавнего времени во многих компаниях, включая Sun, существовали патентные и экспортные ограничения на использование мощных алгоритмов шифрования. К счастью, сегодня все эти ограничения являются менее жесткими, а срок патентного ограничения на использование алгоритма RSA и вовсе истек в октябре 2000 г. Что касается Java SE 1.4, то в стандартной библиотеке этой версии предусмотрены достаточно хорошие средства шифрования.

Симметричные шифры

Криптографическое расширение Java содержит класс `Cipher`, который является суперклассом всех классов, имеющих отношение к шифрованию. Для создания объекта, реализующего алгоритм шифрования, используется метод `getInstance()`:

```
Cipher cipher = Chiper.getInstance(algorithName);
Cipher cipher = Chiper.getInstance(algorithName, providerName);
```

В JDK для всех шифров используется поставщик `SunJCE`. Если имя поставщика не указано явно, то по умолчанию принимается имя `SunJCE`. Если вы хотите воспользоваться алгоритмами, которые не поддерживаются инструментами Sun, следует указать другого поставщика.

Имя алгоритма задается в виде строки, например `"DES"` или `"DES/CBC/PKCS5Padding"`.

Алгоритм DES (Data Encryption Standard — стандарт шифрования данных) — это один из наиболее старых алгоритмов шифрования с длиной ключа в 56 бит. В настоящее время он считается устаревшим, поскольку может взламываться путем при-

менения метода грубой силы (пример можно найти по адресу http://www.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/). Гораздо более эффективным вариантом является появившийся после него алгоритм AES (Advanced Encryption Standard — расширенный стандарт шифрования), подробное описание которого можно найти по адресу http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf. В рассматриваемых здесь примерах мы будем использовать алгоритм AES.

После создания реализующего алгоритм шифрования объекта его нужно инициализировать путем указания режима и ключа (то есть путем установки параметров mode и key):

```
int mode = ...;
Key key = ...;
cipher.init(mode, key);
```

Параметр mode может иметь одно из следующих значений:

```
Cipher.ENCRYPT_MODE
Cipher.DECRYPT_MODE
Cipher.WRAP_MODE
Cipher.UNWRAP_MODE
```

Режимы *свертки* (wrap) и *развертки* (unwrap) применяются для шифрования одного ключа на основе другого. Пример использования таких режимов будет приведен в следующем разделе.

После этого можно повторно вызывать метод update() для выполнения шифрования всех требующихся блоков данных:

```
int blockSize = cipher.getBlockSize();
byte[] inBytes = new byte[blockSize];
. . . // Чтение inBytes.
int outputSize= cipher.getOutputSize(inLength);
byte[] outBytes = new byte[outputSize];
int outLength = cipher.update(inBytes, 0, outputSize, outBytes);
. . . // Запись outBytes.
```

По завершении нужно обязательно один раз вызвать метод doFinal(). Если доступен последний блок входных данных (меньшего, чем указано blockSize, объема), тогда вызов должен выглядеть следующим образом:

```
outBytes = cipher.doFinal(inBytes, 0, inLength);
```

А если были зашифрованы все входные данные, тогда этот вызов должен выглядеть так:

```
outBytes = cipher.doFinal();
```

Вызов метода doFinal() необходим для *дополнения* (padding) заключительного блока данных. Например, возьмем алгоритм шифрования DES. В этом алгоритме размер блока составляет 8 байт. Предположим, что размер последнего блока входных данных оказывается менее 8 байт. Конечно, мы можем дополнить недостающие байты нулями так, чтобы он занимал 8 байт и зашифровать его. Но тогда при расшифровке блоков результат будет содержать в конце несколько присоединенных нулей и, следовательно, немного отличаться от исходного файла входных данных. Это может представлять проблему, и для ее устранения как раз и необходима *схема дополнения* (padding scheme). Одной из наиболее часто применяемых схем дополнения является схема, описанная в документе Public Key Cryptography Standard (PKCS) #5 специалистами из RSA Security Inc. (http://www.rsasecurity.com/rsalabs/pkcs/pkcs-5). В этой схеме последний блок дополняется не нулями, а числами, равными недостающему количеству бай-

тов. Другими словами, если L представляет собой последний (неполный) блок, тогда он будет дополнен 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 входных данных фактически может быть поделена на 8, тогда к входным данным присоединяется и шифруется только один такой блок:

```
08 08 08 08 08 08 08 08
```

При расшифровке на количество подлежащих отсечению дополнительных символов указывает самый последний байт простого текста.

Генерация ключей

Для выполнения шифрования требуется генерировать ключ. Каждый алгоритм шифрования предусматривает свой формат для ключей, но главное, чтобы их генерация выполнялась случайным образом. Чтобы получить сгенерированный полностью случайным образом ключ, потребуется выполнить три перечисленных ниже действия.

1. Создать объект KeyGenerator.
2. Инициализировать генератор случайных чисел. Если длина блока шифра является переменной, тогда нужно будет еще также указать и желаемую длину блока.
3. Вызвать метод generateKey().

Например, генерация ключа AES будет выглядеть следующим образом:

```
KeyGenerator keygen = KeyGenerator.getInstance("AES");
SecureRandom random = new SecureRandom(); //см. ниже
keygen.init(random);
Key key = keygen.generateKey();
```

В качестве альтернативного варианта, ключ еще также можно генерировать и на основании фиксированного набора необработанных данных (например, на основании пароля или вводимых с клавиатуры символов). Тогда нужно воспользоваться классом SecretKeyFactory, как показано ниже:

```
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("AES");
byte[] keyData = . . .; // 16 байтов для AES
SecretKeySpec keySpec = new SecretKeySpec(keyData, "AES");
Key key = keyFactory.generateSecret(keySpec);
```

При генерировании ключей обязательно удостоверяйтесь в том, что используются *по-настоящему случайные* числа. Например, обычный генератор случайных чисел в классе Random, в роли начальных значений (seeds) в котором выступают значения текущей даты и времени, не будет генерировать достаточно случайные числа. Предположим, что часы компьютера показывают время с точностью до 1/10 секунды. Тогда это означает, что на каждый день может приходиться максимум по 864 000 начальных значений. Следовательно, узнав день генерации ключа (который часто очень легко можно вычислить по дате сообщения или дате истечения срока сертификата), злоумышленнику останется просто сгенерировать все возможные значения за этот день.

Класс SecureRandom генерирует гораздо более надежные случайные числа, чем класс Random. Предоставлять ему начальное значение, с которого должна начинаться числовая последовательность в произвольной точке, все равно нужно. Наиболее

удобным способом для выполнения этой операции является получение случайных входных данных с какого-то аппаратного устройства, вроде генератора "белого шума". Другим целесообразным способом для получения случайных входных данных может быть приглашение 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 просто передаваться методу `setSeed`.

```
SecureRandom secrand = new SecureRandom();
byte[] b = new byte[20];
// заполнение по-настоящему случайными битами
secrand.setSeed(b);
```

Если не указывать генератору случайных чисел никакого начального значения, он будет создавать свое собственное 20-байтовое значение путем запуска потоков, перевода их в спящий режим и вычисления точного времени их пробуждения.



О том, является ли такой алгоритм безопасным, не известно. Раньше алгоритмы, действие которых основывалось на вычисленные временных показателей других компонентов компьютера, вроде показателей времени доступа к жесткому диску, очень часто позже оказывались генерирующими недостаточно случайные числа.

Приведенная в листинге 9.17 программа иллюстрирует применение алгоритма шифрования AES. Чтобы использовать ее, сначала нужно сгенерировать секретный ключ:

```
java AESTest -genkey secret.key
```

Полученный секретный ключ следует сохранить в файле `secret.key`.

После этого шифрование можно выполнять с помощью такой команды:

```
java AEStest -encrypt plaintextFile encryptedFile secret.key
```

Расшифровка осуществляется посредством следующей команды:

```
java AEStest -decrypt encryptedFile decryptedFile secret.key
```

Код этой программы выглядит довольно просто. Сначала с помощью опции `-genkey` создается новый секретный ключ и осуществляется его сохранение в указанном файле. Эта операция обычно занимает много времени, поскольку большая его часть уходит на инициализацию генератора случайных чисел. Опции `-encrypt` и `-decrypt` предусматривают вызов одного и того же метода `crypt()`, который, в свою очередь, вызывает такие методы объекта шифрования, как `update()` и `doFinal()`. Важно обратить внимание на то, как метод `update()` вызывается повторно до тех пор, пока блоки входных данных не достигнут полной длины, и на то, что метод `doFinal()` вызывается либо с частично заполненным блоком входных данных (который затем дополняется), либо вообще безо всяких дополнительных данных (для генерации одного устраняющего незаполненное пространство блока).

Листинг 9.17. Содержимое файла `AESTest.java`

```
1.  import java.io.*;
2.  import java.security.*;
3.  import javax.crypto.*;
4.
5.  /**
6.   * Программа проверки алгоритма шифрованния шифра AES.
7.   * Она вызывается следующим образом: <br>
```

```
8.    * java AESTest -genkey keyfile<br>
9.    * java AESTest -encrypt plaintext encrypted keyfile<br>
10.   * java AESTest -decrypt encrypted decrypted keyfile<br>
11.   * @author Cay Horstmann
12.   * @version 1.0 2004-09-14
13.   */
14.  public class AESTest
15.  {
16.    public static void main(String[] args)
17.    {
18.      try
19.      {
20.        if (args[0].equals("-genkey"))
21.        {
22.          KeyGenerator keygen = KeyGenerator.getInstance("AES");
23.          SecureRandom random = new SecureRandom();
24.          keygen.init(random);
25.          SecretKey key = keygen.generateKey();
26.          ObjectOutputStream out = new ObjectOutputStream(new
27.            FileOutputStream(args[1]));
28.          out.writeObject(key);
29.          out.close();
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.          ObjectInputStream keyIn = new ObjectInputStream(new
38.            FileInputStream(args[3]));
39.          Key key = (Key) keyIn.readObject();
40.          keyIn.close();
41.
42.          InputStream in = new FileInputStream(args[1]);
43.          OutputStream out = new FileOutputStream(args[2]);
44.          Cipher cipher = Cipher.getInstance("AES");
45.          cipher.init(mode, key);
46.
47.          crypt(in, out, cipher);
48.          in.close();
49.          out.close();
50.        }
51.      }
52.      catch (IOException e)
53.      {
54.        e.printStackTrace();
55.      }
56.      catch (GeneralSecurityException e)
57.      {
58.        e.printStackTrace();
59.      }
60.      catch (ClassNotFoundException e)
61.      {
62.        e.printStackTrace();
63.      }
64.    }
```

```
65.    /**
66.     * Шифрование данных, полученных из входного потока
67.     * запись зашифрованных байтов в выходной поток.
68.     * @param in Входной поток
69.     * @param out Выходной поток
70.     * @param cipher Шифр
71.     */
72.    public static void crypt(InputStream in,
73.       OutputStream out, Cipher cipher)
74.       throws IOException, GeneralSecurityException
75.    {
76.       int blockSize = cipher.getBlockSize();
77.       int outputSize = cipher.getOutputSize(blockSize);
78.       byte[] inBytes = new byte[blockSize];
79.       byte[] outBytes = new byte[outputSize];
80.
81.       int inLength = 0;
82.       boolean more = true;
83.       while (more)
84.       {
85.          inLength = in.read(inBytes);
86.          if (inLength == blockSize)
87.          {
88.             int outLength = cipher.update(inBytes, 0,
89.                blockSize, outBytes);
90.             out.write(outBytes, 0, outLength);
91.          }
92.          else more = false;
93.       }
94.       if (inLength > 0) outBytes = cipher.doFinal(inBytes, 0, inLength);
95.       else outBytes = cipher.doFinal();
96.       out.write(outBytes);
97.    }
98. }
```

API javax.crypto.Cipher 1.4

- `static Cipher getInstance(String algorithmName)`
- `static Cipher getInstance(String algorithmName, String providerName)`

  Возвращает объект `Cipher`, который реализует указанный алгоритм. Если же таковой не поддерживается, выдает исключение `NoSuchAlgorithmException`.
- `int getBlockSize()`

  Возвращает размер блока в байтах или значение `0`, если алгоритм шифрования не предусматривает возможности работы с блоками.
- `int getOutputSize(int inputLength)`

  Возвращает размер буфера выходных данных, который требуется, если следующие входные данные имеют указанное в `inputLength` количество байтов. Данный метод учитывает любые буферизированные байты в объекте шифрования.
- `void init(int mode, Key key)`

  Инициализирует реализующий алгоритм шифрования объект. Параметр режима может иметь одно из следующих значений: `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)`

  Преобразует один блок входных данных. Первые два метода возвращают выходные данные. Третий метод возвращает информацию о количестве байтов, которые были помещены в массив `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)`

  Преобразует последний блок входных данных и очищает буфер этого объекта алгоритма. Первые три метода возвращают выходные данные, а четвертый метод — информацию о количестве байтов, которые были помещены в массив `out`.

API javax.crypto.KeyGenerator 1.4

- `static KeyGenerator getInstance(String algorithmName)`

  Возвращает объект `KeyGenerator`, который реализует указанный алгоритм. Если таковой не поддерживается, выдает исключение `NoSuchAlgorithmException`.
- `void init(SecureRandom random)`
- `void init(int keySize, SecureRandom random)`

  Инициализирует генератор ключей.
- `SecretKey generateKey()`

  Генерирует новый ключ.

API javax.crypto.SecretKeyFactory 1.4

- `static SecretKeyFactory getInstance(String algorithmName)`
- `static SecretKeyFactory getInstance(String algorithmName, String providerName)`

  Возвращает объект `SecretKeyFactory` для указанного алгоритма.
- `SecretKey generateSecret(KeySpec spec)`

  Создает новый секретный ключ на основе указанной спецификации.

API javax.crypto.spec.SecretKeySpec 1.4

- `SecretKeySpec(byte[] key, String algorithmName)`

  Создает спецификацию ключа.

Потоки шифрования

В библиотеке JCE имеется удобный набор реализующих потоки классов, которые умеют автоматически шифровать и расшифровывать данные потоков. Например, ниже показано, как с помощью одного из таких классов можно шифровать данные и помещать 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); // Получение данных
while (inLength != -1)
{
   out.write(bytes, 0, inLength);
   inLength = getData(bytes); // Получение данных
}
out.flush();
```

Подобным образом для считывания данных из файла и выполнения 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); // помещение данных в пункт назначения
   inLength = in.read(bytes);
}
```

Классы потоков шифрования прозрачным образом обрабатывают вызовы методов `update()` и `doFinal()`, что, несомненно, является важным преимуществом.

API javax.crypto.CipherInputStream 1.4

- `CipherInputStream(InputStream in, Cipher cipher)`

  Создает поток входных данных, который считывает данные из `in` и расшифровывает или зашифровывает их путем использования указанного шифра.
- `int read()`
- `int read(byte[] b, int off, int len)`

  Считывает данные из входного потока и автоматически расшифровывает или зашифровывает их.

API javax.crypto.CipherOutputStream 1.4

- `CipherOutputStream(OutputStream out, Cipher cipher)`

  Создает поток выходных данных, который записывает данные в `out` и зашифровывает или расшифровывает их путем использования указанного шифра.
- `void write(int ch)`
- `void write(byte[] b, int off, int len)`

  Записывает данные в поток выходных данных и автоматически зашифровывает или расшифровывает их.
- `void flush()`

  Принудительно выводит данные из буфера и при необходимости заполняет недостающие биты.

Шифрование с открытым ключом

Алгоритм шифрования DES, рассматривавшийся в предыдущем разделе, является *симметричным*, т.е. подразумевает применение как для шифрования, так и для расшифровки одного и того же ключа. Уязвимым местом всех симметричных алгоритмов шифрования является передача ключа. Например, если Алиса отправляет Бобу зашифрованное сообщение, Бобу для его расшифровки необходим тот же ключ, которым пользовалась Алиса. В случае изменения ключа, Алиса, опять-таки, должна отправить Бобу вместе с сообщением и новую версию этого ключа по какому-то защищенному каналу. Но, возможно, у нее нет такого канала связи с Бобом, и потому она вынуждена шифровать все свои сообщения к нему.

Для решения этой проблемы можно использовать шифрование с 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 просто использовать этот открытый ключ для шифрования всех своих сообщений Бобу.

Однако, к сожалению, все не так уж просто. Дело в том, что все алгоритмы шифрования с открытым ключом работают *намного* медленнее, чем алгоритмы с симметричными ключами вроде DES или AES. Поэтому было бы неэффективно и непрактично использовать открытые ключи для шифрования данных большого объема. Эту проблему легко решить, комбинируя шифрование с открытым ключом и симметричное шифрование. Рассмотрим следующий пример.

1. Алиса создает симметричный ключ случайным образом и с его помощью шифрует свое сообщение.
2. С помощью открытого ключа Боба Алиса зашифровывает симметричный ключ.
3. Алиса посылает Бобу зашифрованное сообщение и зашифрованный симметричный ключ.
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 есть только у него. Таким образом, неэффективный алгоритм шифрования с открытым ключом применяется только для небольшого объема данных симметричного ключа.

Наиболее популярным алгоритмом с открытым ключом является RSA, изобретенный Райвестом, Шамиром и Адлеманом. До октября 2000 г. этот алгоритм был защищен патентом RSA Security Inc., для использования которого требовалось отчислять 3% от прибыли с минимальным ежегодным взносом 50 000 долларов. Сегодня данный алгоритм уже не является коммерческим и поддерживается в Java SE 5.0 и более поздних реализациях.



Тем, кто до сих пор пользуется более старыми версиями JDK, стоит посетить Web-сайт компании Legion of Bouncy Castle (http://www.bouncycastle.org). Эта компания предлагает поставщик, включающий RSA и ряд других алгоритмов, которые в состав SunJCE не входят. Поставщик Legion of Bouncy Castle был сертифицирован Sun Microsystems, так что его можно применять вместе с JDK.

Для использования алгоритма 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, код которой приведен в листинге 9.18, поддерживаются три опции. Опция `-genkey` используется для создания пары ключей, а опция `-encrypt` — для генерации ключа AES и *свертывания* его с помощью открытого ключа.

```
Key key = . . .;          // Ключ AES
Key publicKey = . . .;    // Открытый ключ RSA
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.WRAP_MODE, publicKey);
byte[] wrappedKey = cipher.wrap(key);
```

При этом создается файл, который состоит из таких элементов:

- длина запакованного ключа;
- байты запакованного ключа;
- информация, зашифрованная ключом AES.

Опция `-decrypt` используется для расшифровки файла. Для работы с программой необходимо сначала создать ключи RSA:

```
java RSATest -genKey public.key private.key
```

Затем необходимо зашифровать файл:

```
java RSATest -encrypt plaintextFile encryptedFile public.key
```

И, наконец, нужно расшифровать его и убедиться в том, что полученная информация совпадает с ранее зашифрованной:

```
java RSATest -decrypt encryptedFile decryptedFile private.key
```

Листинг 9.18. Содержимое файла `RSATest.java`

```
1.  import java.io.*;
2.  import java.security.*;
3.  import javax.crypto.*;
4.
5.  /**
6.   * Пример использования алгоритма шифрования RSA.
7.   * Данная программа запускается следующим образом: <br>
8.   * java RSATest -genkey public private<br>
9.   * java RSATest -encrypt plaintext encrypted public<br>
10.  * java RSATest -decrypt encrypted decrypted private<br>
11.  * @author Cay Horstmann
12.  * @version 1.0 2004-09-14
13.  */
14. public class RSATest
15. {
16.    public static void main(String[] args)
17.    {
18.       try
19.       {
```

```
20.         if (args[0].equals("-genkey"))
21.         {
22.            KeyPairGenerator pairgen =
23.               KeyPairGenerator.getInstance("RSA");
24.            SecureRandom random = new SecureRandom();
25.            pairgen.initialize(KEYSIZE, random);
26.            KeyPair keyPair = pairgen.generateKeyPair();
27.            ObjectOutputStream out = new ObjectOutputStream(new
28.               FileOutputStream(args[1]));
29.            out.writeObject(keyPair.getPublic());
30.            out.close();
31.            out = new ObjectOutputStream(new FileOutputStream(args[2]));
32.            out.writeObject(keyPair.getPrivate());
33.            out.close();
34.         }
35.         else if (args[0].equals("-encrypt"))
36.         {
37.            KeyGenerator keygen = KeyGenerator.getInstance("AES");
38.            SecureRandom random = new SecureRandom();
39.            keygen.init(random);
40.            SecretKey key = keygen.generateKey();
41.
42.            // Свертывание с помощью открытого ключа RSA
43.            ObjectInputStream keyIn = new ObjectInputStream(new
44.               FileInputStream(args[3]));
45.            Key publicKey = (Key) keyIn.readObject();
46.            keyIn.close();
47.
48.            Cipher cipher = Cipher.getInstance("RSA");
49.            cipher.init(Cipher.WRAP_MODE, publicKey);
50.            byte[] wrappedKey = cipher.wrap(key);
51.            DataOutputStream out = new DataOutputStream(new
52.               FileOutputStream(args[2]));
53.            out.writeInt(wrappedKey.length);
54.            out.write(wrappedKey);
55.
56.            InputStream in = new FileInputStream(args[1]);
57.            cipher = Cipher.getInstance("AES");
58.            cipher.init(Cipher.ENCRYPT_MODE, key);
59.            crypt(in, out, cipher);
60.            in.close();
61.            out.close();
62.         }
63.         else
64.         {
65.            DataInputStream in = new DataInputStream(new
66.               FileInputStream(args[1]));
67.            int length = in.readInt();
68.            byte[] wrappedKey = new byte[length];
69.            in.read(wrappedKey, 0, length);
70.
71.            // Развертывание с помощью секретного ключа RSA.
72.            ObjectInputStream keyIn = new ObjectInputStream(new
73.               FileInputStream(args[3]));
74.            Key privateKey = (Key) keyIn.readObject();
75.            keyIn.close();
76.            Cipher cipher = Cipher.getInstance("RSA");
77.            cipher.init(Cipher.UNWRAP_MODE, privateKey);
```

```
78.            Key key = cipher.unwrap(wrappedKey, "AES", Cipher.SECRET_KEY);
79.            OutputStream out = new FileOutputStream(args[2]);
80.            cipher = Cipher.getInstance("AES");
81.            cipher.init(Cipher.DECRYPT_MODE, key);
82.            crypt(in, out, cipher);
83.            in.close();
84.            out.close();
85.         }
86.       }
87.       catch (IOException e)
88.       {
89.         e.printStackTrace();
90.       }
91.       catch (GeneralSecurityException e)
92.       {
93.         e.printStackTrace();
94.       }
95.       catch (ClassNotFoundException e)
96.       {
97.         e.printStackTrace();
98.       }
99.    }
100.   /**
101.    * Использование алгоритма шифрования для преобразования байтов
102.    * входного потока и передачи преобразованных байтов
103.    * в выходной поток.
104.    * @param in Входной поток
105.    * @param out Выходной поток
106.    * @param cipher Шифр для преобразования байтов
107.    */
108.   public static void crypt(InputStream in, OutputStream out,
109.      Cipher cipher) throws IOException, GeneralSecurityException
110.   {
111.      int blockSize = cipher.getBlockSize();
112.      int outputSize = cipher.getOutputSize(blockSize);
113.      byte[] inBytes = new byte[blockSize];
114.      byte[] outBytes = new byte[outputSize];
115.
116.      int inLength = 0;
117.      boolean more = true;
118.      while (more)
119.      {
120.         inLength = in.read(inBytes);
121.         if (inLength == blockSize)
122.         {
123.            int outLength = cipher.update(inBytes, 0, blockSize, outBytes);
124.            out.write(outBytes, 0, outLength);
125.         }
126.         else more = false;
127.      }
128.      if (inLength > 0)
129.         outBytes = cipher.doFinal(inBytes, 0, inLength);
130.      else outBytes = cipher.doFinal();
131.      out.write(outBytes);
132.   }
133.   private static final int KEYSIZE = 512;
134. }
```

Итак, в этой главе мы показали, как модель безопасности Java позволяет обеспечивать контролируемое выполнение кода, что является уникальным и чрезвычайно важным аспектом платформы Java. Помимо этого были также рассмотрены различные предлагаемые в библиотеке Java службы для аутентификации и шифрования. Однако существует еще много дополнительных и специфических вопросов, которых мы не коснулись.

- GSS-API-интерфейс для "универсальных служб безопасности", обеспечивающий поддержку для протокола Kerberos (и, в принципе, других протоколов, предназначенных для безопасного обмена сообщениями). Подробнее об этом интерфейсе можно узнать по следующему адресу:

  `http://java.sun.com/javase/6/docs/technotes/guides/security/jgss/tutorials/index.html`

- Поддержка для механизма SASL (Simple Authentication and Security Layer), применяемого протоколами LDAP и IMAP. При необходимости реализовать SASL в своем приложении, всю необходимую информацию можно найти по следующему адресу:

  `http://java.sun.com/javase/6/docs/technotes/guides/security/sasl/sasl-refguide.html`

- Поддержка для SSL (Secure Sockets Layer). Применение SSL через HTTP является вполне понятным для разработчиков приложений и заключается просто в использовании URL-адресов, начинающихся с `https`. При желании добавить SSL в свое собственное приложение, всю необходимую информацию можно найти в документе JSEE (Java Secure Socket Extension) по следующему адресу:

  `http://java.sun.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html`

На этом обзор средств безопасности Java завершается, и в главе 10 мы перейдем к рассмотрению распределенных вычислений.

Глава 10

РАБОТА С УДАЛЕННЫМИ ОБЪЕКТАМИ

В ЭТОЙ ГЛАВЕ...

- Клиент и сервер
- Удаленные вызовы методов
- Модель программирования RMI
- Параметры и возвращаемые значения в удаленных методах
- Активизация удаленных объектов
- Web-службы и JAX-WS

Периодически программисты начинают думать о "повсеместном использовании объектов" как о решении для всех своих проблем. Идея состоит в создании эффективного набора взаимодействующих распределенных объектов, которые могли бы размещаться где угодно. Тогда при необходимости объекта на одном компьютере вызывать метод в объекте на другом компьютере он мог бы отправлять сетевое сообщение со всеми деталями запроса. Удаленный объект затем вычислял бы ответ, к примеру, путем получения доступа к базе данных или обмена данными с дополнительными объектами. Подготовив ответ на запрос клиента, он отсылал бы его обратно, опять-таки, по сети. С теоретической точки зрения этот процесс выглядит довольно просто, но для того, чтобы использовать распределенные объекты эффективно, необходимо понимать, что происходит внутри него.

В этой главе мы расскажем о предлагаемых в Java технологиях для распределенного программирования и в частности о протоколе RMI (Remote Method Invocation — удаленный вызов методов), позволяющем обеспечивать взаимодействие между двумя виртуальными машинами Java (которые могут находиться и на разных компьютерах). Затем мы вкратце рассмотрим технологию JAX-WS, позволяющую осуществлять удаленные вызовы к Web-службам.

Клиент и сервер

Главная идея, лежащая в основе всех технологий распределенного программирования, выглядит довольно просто. Клиентский компьютер создает запрос и отправляет его по сети серверу. Тот обрабатывает запрос и отправляет ответ обратно клиенту для дальнейшего анализа. Весь этот процесс схематично показан на рис. 10.1.

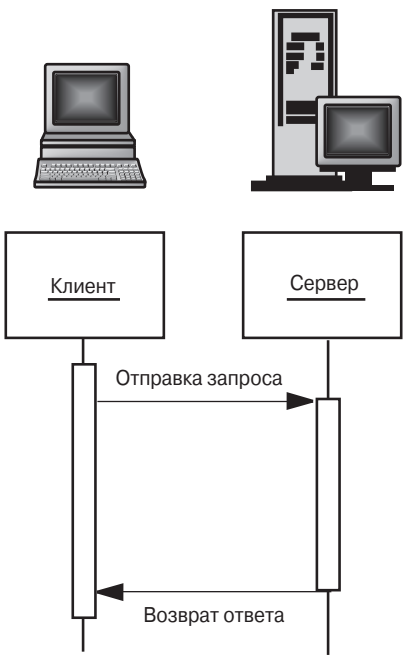


Рис. 10.1. Передача объектов между клиентом и сервером

Вначале хотелось бы отметить, что подобные запросы и ответы не являются такими же, как те, что применяются в Web-приложениях. В роли клиента в данном случае выступает не Web-браузер. Это может быть любое приложение, реализующее бизнес-правила любой сложности. Это клиентское приложение может как предусматривать, так и не предусматривать взаимодействие с пользователем, а если предусматривает, то может иметь как интерфейс командной строки, так и графический интерфейс типа Swing. Применяемый им для передачи запросов и ответов протокол должен поддерживать пересылку произвольных объектов, в то время как в традиционных Web-приложениях для запросов может применяться только HTTP, а для ответов — только HTML.

То есть разработчикам клиентских программ необходим механизм, с помощью которого они могли бы делать обычные вызовы методов, не волнуясь ни о пересылке данных по сети, ни о выполнении синтаксического анализа ответных данных. Для них решением является установка на клиенте так называемого прокси-объекта (proxy object). Прокси-объект — это такой объект, который размещается на виртуальной машине клиента и выглядит для клиентской программы так, будто бы он является удаленным объектом. Клиент может вызывать этот прокси-объект путем выполнения обычного вызова метода, а тот затем уже связываться с сервером посредством сетевого протокола.

Точно так же разработчикам служб необходим механизм, который бы позволял им не беспокоиться об обмене данными с клиентами. Для них решением является установка на сервере второго прокси-объекта. Прокси-объект сервера может взаимодействовать с прокси-объектом клиента и выполнять обычные вызовы методов для обмена данными с объектом, реализующим службу (рис. 10.2).

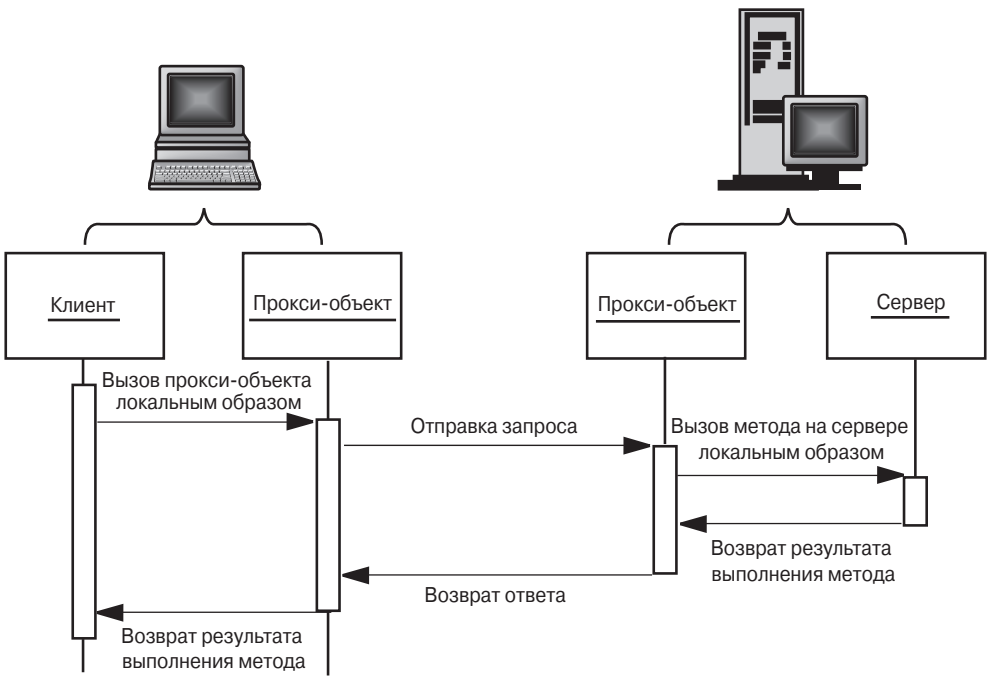


Рис. 10.2. Удаленный вызов методов с использованием прокси-объектов

Каким образом прокси-объекты могут взаимодействовать друг с другом? Это зависит от применяемой для их реализации технологии. Наиболее распространенными являются три следующих варианта.

- Технология Java RMI (Remote Method Invocation — удаленный вызов методов), которая позволяет обеспечивать вызов методов между распределенными объектами Java.
- Технология CORBA (Common Object Request Broker Architecture — общая архитектура посредника запросов к объектам), которая позволяет обеспечивать вызов методов между объектами, написанными на любом языке программирования. Взаимодействие между объектами в этой технологии реализуется на основе протокола IIOP (Internet Inter-ORB Protocol).
- Архитектура Web-служб, которая представляет собой целую коллекцию протоколов, часто в общем описываемых с помощью префикса WS-*. Она тоже подходит для объектов, написанных на любом языке программирования, однако для обеспечения взаимодействия между ними предусматривает применение форматов, основанных на XML. Форматом для передачи объектов является SOAP (Simple Object Access Protocol — простой протокол доступа к объектам).

В случае реализации рассчитанных на взаимодействие программ с помощью кода Java, в универсальности и сложности CORBA и WS-* нет абсолютно никакой необходимости. Такой простой механизм, как RMI, был специально разработан компанией Sun для обеспечения возможности взаимодействия между приложениями Java.

Познакомиться с технологией RMI стоит даже тем, кто и не планирует применять его в своих собственных программах. Это позволит изучить все важные для программирования распределенных приложений механизмы на примере довольно простой архитектуры. Более того, тем, кто пользуется Java-технологиями уровня предприятия, тоже будет очень полезно получить хотя бы общее представление о RMI, поскольку там именно этот протокол и применяется для обеспечения взаимодействия между bean-компонентами уровня предприятия (Enterprise Java Beans — EJB). EJB-компоненты — это применяемые на стороне сервера компоненты, которые создаются для образования сложных приложений, работающих на множестве серверов. Дабы эффективно их использовать, потребуется хорошо разбираться в сопряженных с удаленными вызовами затратах.

В отличие от RMI, технологии CORBA и WS-* не требуют использования никакого конкретного языка программирования. Они позволяют создавать клиентские и серверные программы на C, C++, Java и любом другом языке программирования. От разработчика ожидается предоставление *описания интерфейса* с указанием сигнатур методов и типов данных, которые его объекты способны обрабатывать. Это описание должно составляться на специальном языке, называемом IDL (Interface Definition Language — язык описания интерфейсов), если речь идет о CORBA, или WSDL (Web Services Description Language — язык описания Web-служб), если речь идет о WS-*.

На протяжении многих лет довольно много людей считали CORBA объектной моделью будущего. Однако сегодня, откровенно говоря, CORBA имеет репутацию (порой вполне заслуженную) архитектуры с низкой производительностью, сложной реализацией и проблемами взаимодействия, потому не пользуется особым успехом. Мы рассказывали о возможностях взаимодействия между Java и CORBA в предыдущих изданиях этой книги, но в этом издании решили уже больше не касаться этой темы, поскольку она перестала быть актуальной.

Web-службы тоже считались не менее перспективными, когда только появились, поскольку имели более простой вид и, конечно же, основывались на эффективных

возможностях WWW и XML. Однако по прошествии времени после доработок многих комитетов стек протоколов Web-служб стал выглядеть гораздо сложнее и включать многие из тех же возможностей, которые уже давно присутствовали в CORBA. Протокол на базе XML хорош тем, что делает данные (достаточно) понятными для человека и тем самым упрощает процесс отладки. Однако с другой стороны, обработка XML-данных существенно снижает производительность. Недавно стек WS-* утратил довольно приличную долю своей привлекательности и стал тоже обзаводиться (порой вполне заслуженно) репутацией сложной для реализации технологии с массой проблем в области обеспечения взаимодействия.

В конце этой главы мы приведем пример приложения, пользующегося Web-службой, и рассмотрим применяемый для этого протокол, дабы продемонстрировать, как может обеспечиваться возможность взаимодействия между различными языками программирования.

Удаленные вызовы методов

Ключевым понятием в области распределенной обработки данных является *удаленный вызов метода* (remote method call). Предположим, что некий код на одной машине (называемый *клиентом*) хочет вызвать метод для объекта на другой машине (называемого *удаленным объектом*). Дабы подобное стало возможным, ему нужно как-то переслать параметры этого метода на другую машину, указать серверу на необходимость отыскать удаленный объект и выполнить для него этот метод, а затем как-то вернуть полученное значение обратно на исходную машину.

Прежде чем подробно рассматривать этот процесс, предлагаем обратить внимание на то, что терминология "клиент-сервер" касается только одного единственного вызова метода. Компьютер, вызывающий удаленный метод, выступает для *этого* вызова в роли клиента, а компьютер, на котором находится требуемый для обработки этого вызова объект — в роли сервера. При выполнении следующей операции эти компьютеры запросто могут поменяться ролями, то есть сервер предыдущего вызова при удаленном вызове метода для объекта, находящегося на другом компьютере, может легко сам стать клиентом.

Заглушки и передача параметров

При необходимости вызвать метод на удаленном объекте, клиентский код на самом деле вызывает обычный метод на прокси-объекте, называемом *объектом-заглушкой* (stub). Например:

```
Warehouse centralWarehouse = get stub object;
double price = centralWarehouse.getPrice("Blackwell Toaster");
```

Объект-заглушка находится на клиентском компьютере, а не на сервере, и просто знает, как можно связаться с сервером по сети. Он упаковывает параметры удаленного метода в блок байтов. Процесс кодирования параметров называется *маршализацией параметров* (parameter marshalling). Его главной целью является преобразование параметров в формат, пригодный для их передачи с одной виртуальной машины на другую. В случае применения протокола RMI объекты кодируются с помощью механизма сериализации, о котором рассказывалось в главе 1, а в случае применения протокола SOAP они кодируются как XML.

В общем, вызов метода объекта-заглушки на клиентском компьютере приводит к созданию информационного блока, содержащего следующие элементы:

- идентификатор целевого удаленного объекта;
- описание подлежащего вызыванию метода;
- параметры.

Далее объект-заглушка отправляет эту информацию серверу. Затем, на стороне сервера объект-получатель выполняет с ее помощью следующие действия.

1. Отыскивает требуемый объект.
2. Вызывает необходимый метод, передавая ему указанные параметры.
3. Перехватывает возвращаемое значение или сгенерированное при вызове данного метода исключение.
4. Отправляет пакет с маршализированными возвращенными данными обратно находящемуся на клиентском компьютере объекту-заглушке.

После этого клиентский объект-заглушка демаршализирует возвращенное с сервера значение или исключение. Значение становится возвращаемым значением вызова метода самого объекта-заглушки, а исключение просто выдается повторно в виртуальной машине вызывавшей программы. На рис. 10.3 схематично показано, как выглядит весь этот процесс.

Данный процесс явно является сложным, но хорошая новость состоит в том, что происходит он полностью автоматически и в большой степени является прозрачным для программиста.

То, как будут выглядеть детали реализации удаленных объектов и получения клиентских объектов-заглушек, зависит от применяемой для распределенных объектов технологии. В следующих разделах довольно подробно рассматривается применение технологии RMI.

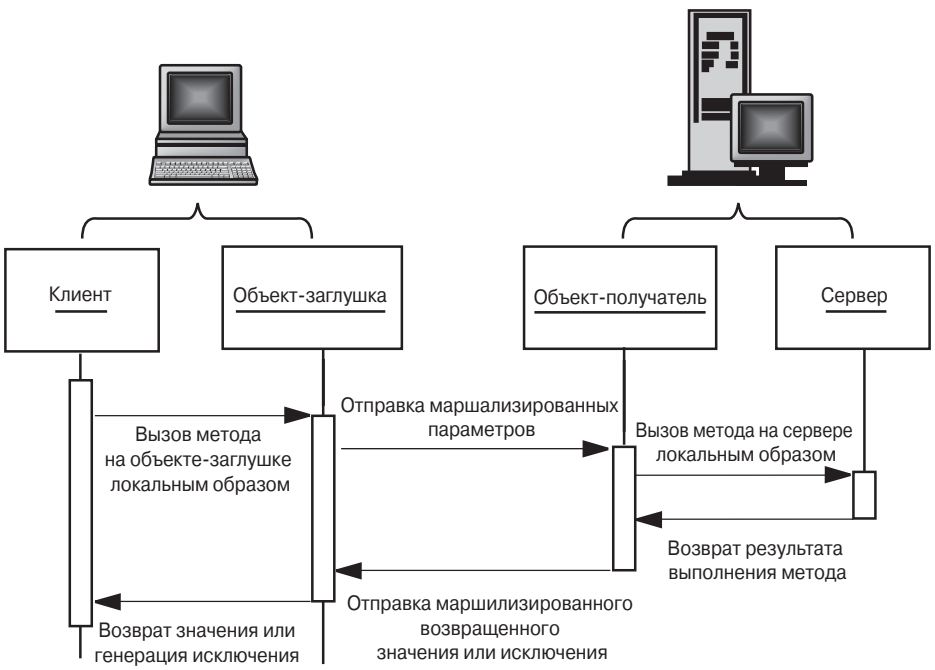


Рис. 10.3. Маршализация параметров

Модель программирования RMI

Для описания модели программирования RMI мы будем использовать простой пример, в котором в роли удаленного объекта будет выступать хранилище данных о различных товарах, а клиентская программа будет запрашивать у него информацию о цене того или иного продукта. В следующих разделах мы покажем, как можно реализовать и запустить в действие необходимую для этого примера серверную и клиентскую программу.

Интерфейсы и реализации

Возможности удаленных объектов выражаются в интерфейсах, которые разделятся между клиентом и сервером. Например, интерфейс, код которого приведен в листинге 10.1, описывает службу, предоставляемую удаленным объектом хранилища данных.

Листинг 10.1. Содержимое файла `Warehouse.java`

```
1.  import java.rmi.*;
2.
3.  /**
4.   * Удаленный интерфейс для простого хранилища данных.
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. }
```

Интерфейсы удаленных объектов должны всегда представлять собой расширение поставляемого в пакете `java.rmi` интерфейса `Remote`. Все методы в этих интерфейсах также должны обязательно объявлять о том, что они могут выдавать исключение `RemoteException`. Дело в том, что удаленные вызовы методов по своей природе являются не такими надежными, как вызовы локальных методов, а это значит, что всегда существует вероятность завершения удаленного вызова неудачей. Например, сервер может оказаться временно недоступным, или в сети могут возникнуть какие-то проблемы. Клиентский код должен быть всегда готов к обработке подобных ситуаций. Именно поэтому при *каждом* вызове удаленного метода и требуется перехватывать исключение `RemoteException` и указывать подлежащее выполнению в случае неудачного завершения вызова действие.

Далее на стороне сервера необходимо создать класс, который в действительности будет выполнять всю указанную в удаленном интерфейсе работу, как показано в листинге 10.2.

Листинг 10.2. Содержимое файла `WarehouseImpl.java`

```
1.  import java.rmi.*;
2.  import java.rmi.server.*;
3.  import java.util.*;
4.
5.  /**
6.   * Данный класс является реализацией удаленного интерфейса Warehouse.
7.   * @version 1.0 2007-10-09
```

```
8.    * @author Cay Horstmann
9.    */
10.  public class WarehouseImpl extends UnicastRemoteObject implements Warehouse
11.  {
12.    public WarehouseImpl() throws RemoteException
13.    {
14.      prices = new HashMap<String, Double>();
15.      prices.put("Blackwell Toaster", 24.95);
16.      prices.put("ZapXpress Microwave Oven", 49.95);
17.    }
18.
19.    public double getPrice(String description) throws RemoteException
20.    {
21.      Double price = prices.get(description);
22.      return price == null ? 0 : price;
23.    }
24.    private Map<String, Double> prices;
25.  }
```



В конструкторе `WarehouseImpl` указывается, что он должен выдавать исключение `RemoteException`, поскольку конструктор суперкласса тоже может выдавать такое исключение. Это происходит при возникновении проблем с подключением к службе, отвечающей за отслеживание удаленных объектов.

Можно сказать, что данный класс выступает в роли целевого объекта для удаленных вызовов методов, поскольку он является расширением класса `UnicastRemoteObject`. Конструктор этого класса делает объекты доступными удаленно. Наследование от класса `UnicastRemoteObject` является наиболее простым "путем", и поэтому все демонстрируемые в этой главе классы реализации служб наследуются именно от него.

В некоторых случаях наследование от класса `UnicastRemoteObject` может оказаться нежелательным, например, из-за того, что класс реализации уже наследуется от другого класса. В таких случаях необходимо вручную создавать экземпляры удаленных объектов и передавать их статическому методу `exportObject()`. Вместо расширения класса `UnicastRemoteObject` в конструкторе удаленного объекта потребуется делать следующий вызов:

```
UnicastRemoteObject.exportObject(this, 0);
```

Второй параметр `0` указывает, что для прослушивания клиентских подключений может использоваться любой подходящий порт.



Термин *unicast* (одноадресный) означает, что удаленный объект должен обнаруживаться путем обращения к одному IP-адресу и порту. Данный механизм является единственным, который поддерживается в Java SE. Более сложные системы распределенных объектов вроде JINI) позволяют выполнять многоадресный (multicast) поиск удаленных объектов, которые могут находиться на множестве различных серверов.

Служба регистрации RMI

Для получения доступа к находящемуся на сервере удаленному объекту, клиенту необходим локальный объект-заглушка. Как же клиент может запрашивать такой объект? Наиболее распространенным способом для этого является вызов удаленного метода

другого удаленного объекта и получение объекта-заглушки в виде возвращаемого значения. При таком подходе, однако, возникает проблема "курицы и яйца": ведь тогда еще требуется каким-то образом получать доступ к самому *первому* удаленному объекту. Для этой цели JDK предлагается *служба реестра самозагрузки* (bootstrap registry service).

Серверная программа регистрирует с помощью этой службы как минимум один удаленный объект. Для регистрации удаленного объекта требуется всего лишь URL-адрес RMI и ссылка на объект реализации.

URL-адрес RMI начинается с префикса `rmi:` и содержит необязательное имя хоста, необязательный номер порта и имя удаленного объекта, которое (что очень желательно) является уникальным. Например:

```
rmi://regserver.mycompany.com:99/central_warehouse
```

По умолчанию именем хоста является `localhost`, а номером порта — `1099`. Сервер указывает службе регистрации в заданном месте ассоциировать, или "связать", имя с объектом.

Ниже приведен код, необходимый для регистрации объекта `WarehouseImpl` с помощью службы регистрации RMI на том же сервере:

```
WarehouseImpl centralWarehouse = new WarehouseImpl();
Context namingContext = new InitialContext();
namingContext.bind("rmi:central_warehouse", centralWarehouse);
```

Программа, показанная в листинге 10.3, просто создает и регистрирует объект `WarehouseImpl`.

Листинг 10.3. Содержимое файла `WarehouseServer.java`

```
1.  import java.rmi.*;
2.  import javax.naming.*;
3.
4.  /**
5.   * Эта серверная программа создает экземпляр удаленного объекта Warehouse,
6.   * регистрирует его с помощью службы имен и ожидает, когда клиенты начнут
7.   * вызывать методы.
8.   * @version 1.12 2007-10-09
9.   * @author Cay Horstmann
10.  */
11. public class WarehouseServer
12. {
13.   public static void main(String[] args) throws RemoteException, NamingException
14.   {
15.     System.out.println("Constructing server implementation...");
16.                   // Создание серверной реализации.
17.     WarehouseImpl centralWarehouse = new WarehouseImpl();
18.
19.     System.out.println("Binding server implementation to registry...");
20.                  // Связывание серверной реализации с реестром.
21.     Context namingContext = new InitialContext();
22.     namingContext.bind("rmi:central_warehouse", centralWarehouse);
23.
24.     System.out.println("Waiting for invocations from clients...");
25.                   // Ожидание вызовов от клиентов.
26.   }
27. }
```



Из соображений безопасности приложение может связывать, удалять связывание или заново связывать ссылки объектов реестра только в том случае, если оно находится на том же компьютере, что и реестр. Это исключает вероятность внесения изменений в информацию реестра злонамеренными клиентами. Однако просматривать объекты все равно может любой клиент.

Клиент может перечислять все зарегистрированные объекты RMI путем вызова следующего метода:

```
Enumeration<NameClassPair> e = namingContext.list("rmi://regserver.mycompany.com");
```

`NameClassPair` — это вспомогательный класс, в котором содержится как имя самого связанного объекта, так и имя его класса. Например, с помощью следующего кода можно отобразить имена всех зарегистрированных объектов:

```
while (e.hasMoreElements())
  System.out.println(e.nextElement().getName());
```

Клиент получает необходимый для осуществления доступа к удаленному объекту объект-заглушку путем указания сервера и имени удаленного объекта следующим образом:

```
String url = "rmi://regserver.mycompany.com/central_warehouse";
Warehouse centralWarehouse = (Warehouse) namingContext.lookup(url);
```



Поскольку широко известно, что поддерживать уникальность имен в глобальном реестре очень трудно, пользоваться этим приемом как стандартным методом для определения местонахождения объектов на сервере не стоит. Вместо этого необходимо стараться регистрировать с помощью службы самозагрузки относительно небольшое количество удаленных объектов, причем таких, которые смогут сами устанавливать местонахождение других объектов.

Код, приведенный в листинге 10.4, демонстрирует клиент, который получает объект-заглушку для осуществления доступа к удаленному объекту хранилища данных (`Warehouse`) и вызывает удаленный метод `getPrice()`. На рис. 10.4 схематично показана происходящая при этом передача управления. Клиент получает объект-заглушку `Warehouse` и вызывает на нем метод `getPrice`. Далее этот объект-заглушка незаметным образом связывается с сервером и вынуждает его вызывать метод `getPrice` на объекте `WarehouseImpl`.

Листинг 10.4. Содержимое файла `WarehouseClient.java`

```
1.  import java.rmi.*;
2.  import java.util.*;
3.  import javax.naming.*;
4.  /**
5.   * Клиент, который вызывает удаленный метод.
6.   * @version 1.0 2007-10-09
7.   * @author Cay Horstmann
8.   */
9.  public class WarehouseClient
10. {
11.   public static void main(String[] args) throws NamingException, RemoteException
12.   {
13.     Context namingContext = new InitialContext();
14.
15.     System.out.print("RMI registry bindings: ");  // привязки в реестре RMI
16.     Enumeration<NameClassPair> e = namingContext.list("rmi://localhost/");
```

```
17.     while (e.hasMoreElements())
18.       System.out.println(e.nextElement().getName());
19.
20.     String url = "rmi://localhost/central_warehouse";
21.     Warehouse centralWarehouse = (Warehouse) namingContext.lookup(url);
22.     String descr = "Blackwell Toaster";
23.     double price = centralWarehouse.getPrice(descr);
24.     System.out.println(descr + ": " + price);
25.   }
26. }
```

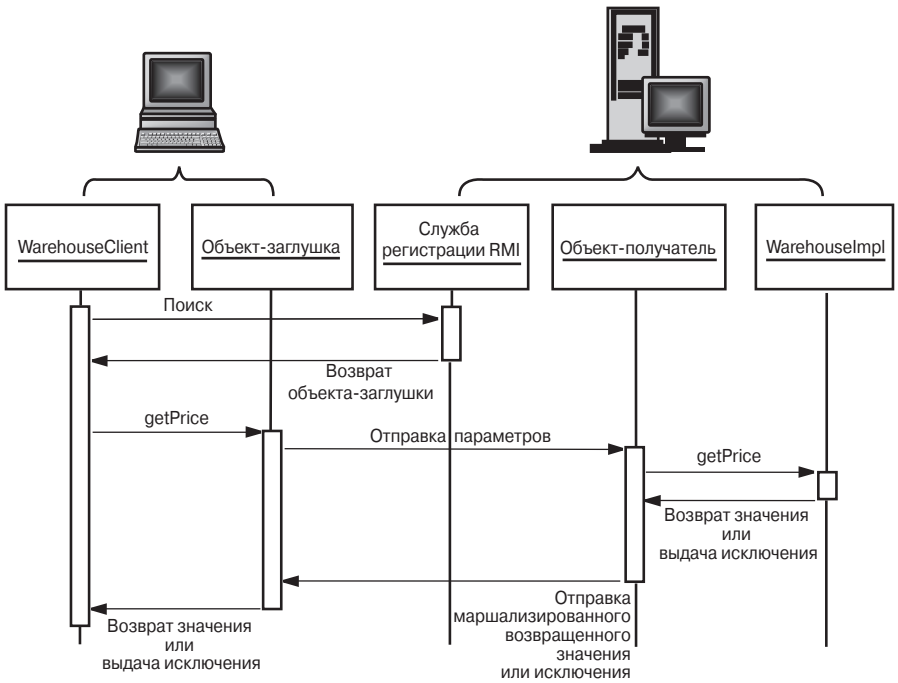


Рис. 10.4. Вызов удаленного метода `getPrice`

API javax.naming.InitialContext 1.3

- `InitialContext()`

  Создает контекст именования, который может использоваться для получения доступа к службе регистрации (реестру) RMI.

API javax.naming.Context 1.3

- `static Object lookup(String name)`

  Возвращает объект, соответствующий указанному имени. Если указанное имя на текущий момент не связано ни с каким объектом, выдает исключение `NamingException`.

- `static void bind(String name, Object obj)`
  Связывает имя `name` с объектом `obj`. Если с данным объектом уже связано какое-то имя, выдает исключение `NameAlreadyBoundException`.
- `static void unbind(String name)`
  Отменяет связь объекта с именем. Позволяет отменять связь и с несуществующим именем.
- `static void rebind(String name, Object obj)`
  Связывает имя `name` с объектом `obj`. Существующая связь заменяется новой.
- `NamingEnumeration<NameClassPair> list(String name)`
  Возвращает перечисление с перечнем всех подходящих связанных объектов. Для получения списка всех RMI-объектов при вызове следует указывать строку `rmi:`.

API javax.naming.NameClassPair 1.3

- `String getName()`
  Получает имя именованного объекта.
- `String getClassName()`
  Получает имя класса, которому принадлежит именованный объект.

API java.rmi.Naming 1.1

- `static Remote lookup(String url)`
  Возвращает удаленный объект для заданного URL. Если указанное имя на текущий момент не связано ни с каким объектом, выдает исключение `NotBoundException`.
- `static void bind(String name, Remote obj)`
  Связывает имя `name` с удаленным объектом `obj`. Если с указанным объектом уже связано какое-то имя, выдает исключение `AlreadyBoundException`.
- `static void unbind(String name)`
  Удаляет связь с именем `name`. Если это имя на текущий момент не связано ни с каким объектом, выдает исключение `NotBoundException`.
- `static void rebind(String name, Remote obj)`
  Связывает имя `name` с удаленным объектом `obj`. Если с этим объектом уже связано какое-нибудь имя, заменяет его старую связь новой.
- `static String[] list(String url)`
  Возвращает массив строк с URL из реестра, расположенного по заданному URL. В этом массиве содержится снимок всех представленных в реестре имен.

Развертывание программы

Развертывание приложения, использующего RMI, может оказаться непростой задачей, поскольку столько всего может пойти не так, а отображаемые при этом сообщения об ошибках могут быть непонятными. Мы пришли к выводу, что процесс развертывания лучше действительно тестировать в реальных условиях, разделяя классы для клиента и для сервера.

Поэтому давайте создадим два отдельных каталога для хранения отвечающих за запуск сервера и клиента классов:

```
server/
  WarehouseServer.class
  Warehouse.class
  WarehouseImpl.class
client/
  WarehouseClient.class
  Warehouse.class
```

При развертывании приложений RMI обычно требуется динамическим образом доставлять классы выполняющимся программам. Одним из примеров тому является реестр RMI. Следует помнить о том, что один экземпляр реестра будет обслуживать множество различных приложений RMI. А это значит, что реестру RMI необходимо иметь доступ к файлам классов всех интерфейсов служб, которые будут регистрироваться. При запуске службы реестра, однако, никто не может предсказать, как будут выглядеть будущие запросы на регистрацию. Поэтому реестру RMI и приходится динамическим образом загружать файлы классов тех удаленных интерфейсов, которые ему раньше не встречались.

Доставленные динамическим образом файлы классов распространяются по стандартным Web-серверам. В нашем случае серверной программе необходимо сделать доступным для реестра RMI файл `Warehouse.class`, поэтому мы помещаем этот файл в третий каталог под названием `download`:

```
download/
  Warehouse.class
```

Для обслуживания содержимого этого каталога применяется Web-сервер.

При развертывании приложения, сервер, реестр RMI, Web-сервер и клиент могут располагаться и на четырех совершенно разных компьютерах, как показано на рис. 10.5. Однако для целей тестирования мы будем использовать только один компьютер.

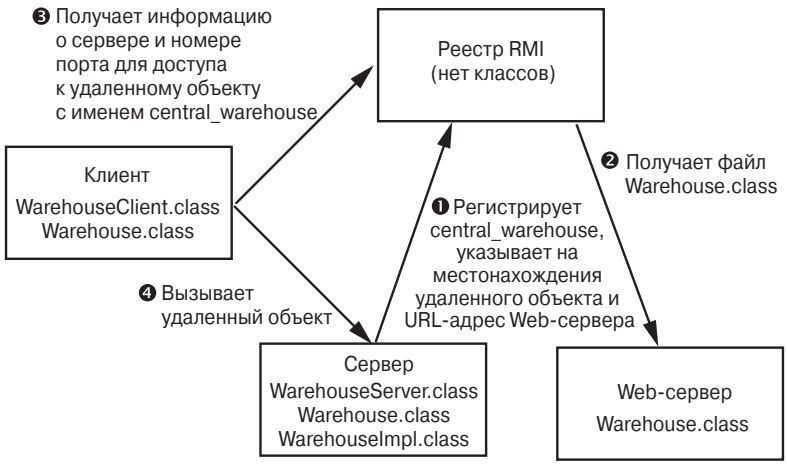


Рис. 10.5. Вызовы сервера в приложении `Warehouse`



Для безопасности службы `rmiregistry`, входящая в состав пакета JDK, позволяет выполнять предназначенные для связывания вызовы только с одного и того же хоста. Другими словами, это означает, что процесс сервера и службы `rmiregistry` должны находиться на одном и том же компьютере. Архитектура RMI, однако, допускает применение и более универсальных реализаций реестра RMI, поддерживающих участие множества серверов.

Чтобы протестировать данное демонстрационное приложение, воспользуйтесь Web-сервером `NanoHTTPD`, который доступен по адресу `http://elonen.iki.fi/code/nanohttpd`. Этот крошечный Web-сервер реализован в виде одного единственного файла с исходным кодом на Java. Откройте новое окно консоли, перейдите в каталог `download` и скопируйте в него файл `NanoHTTPD.java`. Далее скомпилируйте файл исходного кода и запустите Web-сервер с помощью следующей команды:

```
java NanoHTTPD 8080
```

В качестве аргумента требуется указать номер порта. Если порт 8080 уже используется на вашей машине, укажите номер любого другого доступного порта.

Далее откройте еще одно окно консоли, перейдите в каталог, в котором *нет никаких файлов классов*, и запустите службу реестра RMI:

```
Rmiregistry
```



Прежде чем запускать службу реестра RMI обязательно удостоверьтесь в том, что для переменной среды `CLASSPATH` не установлено никакое значение и что в текущем каталоге действительно нет никаких файлов классов. Если это не так, служба реестра RMI может обнаружить какие-нибудь поддельные файлы классов, что будет сбивать ее с толку при необходимости загрузить дополнительные классы из другого источника. Подобное поведение имеет свою причину, подробнее о которой можно узнать по адресу `http://java.sun.com/javase/6/docs/technotes/guides/rmi/codebase.html`. Если объяснять вкратце, то у каждого объекта-заглушки имеется запись о кодовой базе, указывающая на то, откуда он был загружен. Эта запись применяется для загрузки всех зависимых классов. Это означает, что в случае обнаружения класса локальным образом служба реестра RMI будет неправильно определять кодовую базу.

Теперь вы готовы 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-адрес, который должен использоваться для обслуживания файлов классов. Серверная программа передает этот URL-адрес службе реестра RMI.

Загляните на минутку в то окно консоли, в котором выполняется `NanoHTTPD`. Там вы увидите сообщение, показывающее, что файл `Warehouse.class` был передан службе реестра RMI.



Очень важно, чтобы вы удостоверились в том, что URL-адрес кодовой базы *оканчивается символом косой черты* (`/`).

Обратите внимание на то, что серверная программа не завершает работы. Это выглядит странно — ведь получается, что она просто создает объект `WarehouseImpl` и регистрирует его. На самом деле метод `main` все-таки завершает работу сразу же после регистрации, как и следовало ожидать. Однако, при создании объекта класса, расши-

ряющего класс `UnicastRemoteObject`, запускается отдельный поток, удерживающий программу в активном состоянии до бесконечности. Благодаря этому программа продолжает оставаться доступной и позволяет клиентам к ней подключаться.

И, наконец, напоследок откройте еще четвертое окно консоли, перейдите в каталог `client` и выполните следующую команду:

```
java WarehouseClient
```

После этого появится короткое сообщение, показывающее, что удаленный метод был успешно вызван (рис. 10.6).



Желающие протестировать лишь базовую логику программы могут поместить файлы классов клиента и сервера в один и тот же каталог и затем запускать службу реестра RMI, сервер и клиент в этом каталоге. Поскольку загрузка класса RMI является источником многих трудностей и путаницы, мы решили, что будет лучше сразу продемонстрировать правильную процедуру настройки динамической загрузки классов.

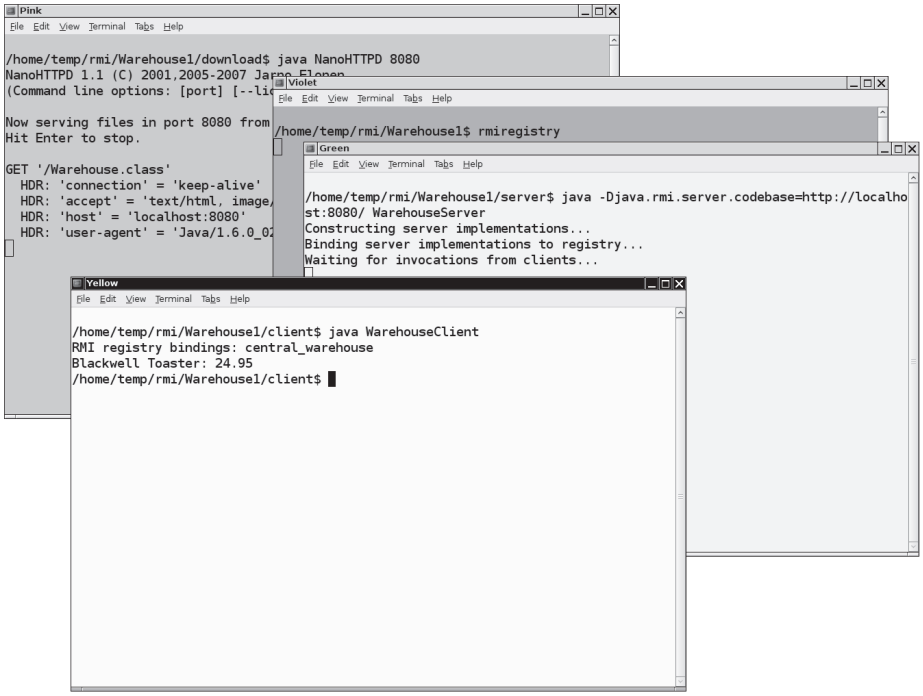


Рис. 10.6. Тестирования приложения RMI

Регистрация активности RMI

В случае запуска сервера с помощью опции

```
-Djava.rmi.server.logCalls=true WarehouseServer &
```

он будет отображать все вызовы удаленных методов в своем окне консоли. Попробуйте сделать так, и вы сможете получить хорошее представление о том, как выглядит трафик RMI.

При желании видеть еще также и все дополнительные касающиеся регистрации сообщения, вам придется сконфигурировать соответствующим образом средства регистрации RMI (loggers) с помощью стандартного API-интерфейса регистрации Java. (Более подробную информацию о регистрации можно найти в главе 11 первого тома.) Чтобы сделать это, 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 `FINE`, вы можете выполнить более точную настройку и установить отдельный уровень для каждого регистратора. Например, для отслеживания действий по загрузке классов вы можете использовать такой параметр настройки:

```
sun.rmi.loader.level=FINE
```

В табл. 10.1 приведен список всех доступных для настройки регистраторов RMI.

Таблица 10.1. Регистраторы RMI

Имя регистратора	Регистрируемые действия
`sun.rmi.server.call`	Удаленные вызовы на стороне сервера
`sun.rmi.server.ref`	Передача удаленной ссылки на стороне сервера
`sun.rmi.client.call`	Удаленные вызовы на стороне клиента
`sun.rmi.client.ref`	Передача удаленной ссылки на стороне клиента
`sun.rmi.dgc`	Распределенная сборка мусора
`sun.rmi.loader`	Действия загрузчика классов RMI (`RMIClassLoader`)
`sun.rmi.transport.misc`	Действия, происходящие на транспортном уровне
`sun.rmi.transport.tcp`	Связывание и подключение по протоколу TCP
`sun.rmi.transport.proxy`	HTTP-туннелирование

Далее запустите службу реестра RMI с использованием следующей опции:

```
-J-Djava.util.logging.config.file=каталог/logging.properties
```

А потом запустите клиент и сервер с использованием такой опции:

```
-Djava.util.logging.config.file=directory/logging.properties
```

Ниже приведен пример регистрационного сообщения, информирующего о возникновении проблемы при загрузке класса. В данном случае оно информирует о том, что службе реестра RMI не удалось отыскать класс `Warehouse` из-за завершения работы Web-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 передаваться параметры, а по завершении вызова в обратном направлении должно, соответственно, передаваться уже возвращаемое значение. При передаче значения от одной виртуальной машины другой мы выделяем два случая: передачу удаленных объектов и передачу не удаленных объектов. Например, предположим, клиент `WarehouseServer` может передавать ссылку `Warehouse` (то есть объект-заглушку, позволяющий вызывать удаленный объект хранилища данных), указывающую на еще один удаленный метод. Это будет примером передачи удаленного объекта. Однако чаще всего в параметрах методов все-таки передаются обычные объекты Java, а не объекты-заглушки, предоставляющие доступ к другим удаленным объектам. Примером тому может служить приводившийся в первом демонстрационном приложении параметр `String` метода `getPrice`.

Передача удаленных объектов

В случае передачи от одной виртуальной машины другой ссылки на удаленный объект и получатель, и отправитель удаленного объекта имеют ссылку на один и тот же логический элемент (entity). Эта ссылка не указывает ни на какое место размещения в памяти (что имело бы смысл только в случае участия одной единственной машины), а состоит из сетевого адреса и уникального идентификатора удаленного объекта. Вся эта информация инкапсулируются в объекте-заглушке.

С концептуальной точки зрения передача удаленной ссылки очень похожа на передачу локальной ссылки в пределах виртуальной машины. Однако всегда следует помнить о том, что вызов метода с использованием удаленной ссылки выполняется гораздо медленнее и потенциально является менее надежным, чем вызов метода с использованием локальной ссылки.

Передача не удаленных объектов

Возьмем параметр `String` из метода `getPrice`. Строковое значение должно копироваться с клиента на сервер. Представить, как копия строки может транспортироваться по сети, совсем не трудно. Механизм RMI может делать копии и даже более сложных объектов, главное, чтобы они были *сериализируемыми* (serializable). Для отправки объектов по сетевому соединению RMI использует механизм сериализации, о котором более подробно рассказывалось в главе 1. Это означает, что в качестве параметров и возвращаемых типов могут применяться любые классы, которые реализуют интерфейс `Serializable`.

Передача параметров за счет их сериализации определенным образом отражается на семантике удаленных методов. В частности, при передаче объектов в локальный метод, в действительности передаются *ссылки* на эти объекты. В случае применения удаленным методом метода мутатора (mutator) к обычному объекту параметра вызывающая программа заметит это, а в случае мутации сериализованного параметра изменениям подвергнется только копия, и вызывающая программа ничего не заметит.

В общем, для передачи значений между виртуальными машинами используются два механизма:

- объекты классов, которые реализуют интерфейс `Remote`, передаются в виде удаленных ссылок;
- объекты классов, которые реализуют интерфейс `Serializable` (а не `Remote`), копируются с применением сериализации.

Все это происходит автоматически и не требует никакого вмешательства со стороны программиста. Однако нужно помнить о том, что в случае больших объектов сериализация может выполняться очень медленно и что удаленный метод не может осуществлять мутацию сериализованных параметров. Этих проблем, конечно, можно избегать путем передачи повсюду удаленных ссылок. Такой подход, однако, тоже имеет свою цену: вызов методов по удаленным ссылкам обходится гораздо дороже вызова локальных методов. Знание о таких затратах позволяет делать осознанный выбор при проектировании удаленных служб.



Удаленные объекты обрабатываются сборщиком мусора автоматически, то есть точно так же, как и локальные объекты. С распределенным сборщиком мусора дело, однако, обстоит, гораздо сложнее. Когда локальный сборщик мусора обнаруживает, что имеются дальнейшие локальные места применения удаленной ссылки, он уведомляет распределенного сборщика о том, что данный клиент уже больше не ссылается на сервер. А когда сервер уже больше не используется никакими клиентами, он помечается как мусор.

Как именно можно передавать удаленные и сериализируемые объекты мы продемонстрируем на примере следующей программы. Для этого мы изменим интерфейс `Warehouse` так, как показано в листинге 10.5, чтобы, получая список ключевых слов, он возвращал наиболее подходящий объект `Product`.

Листинг 10.5. Содержимое файла `Warehouse.java`

```
1.  import java.rmi.*;
2.  import java.rmi.*;
3.  import java.util.*;
4.  /**
5.   * Удаленный интерфейс для простого хранилища данных.
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` имеет тип `List<String>`. Значение этого параметра должно принадлежать классу, поддающемуся сериализации и реализующему интерфейс `List<String>`, такому как, например, класс `ArrayList<String>`. (В нашем примере клиент передает значение, получаемое путем вызова `Arrays.asList`. К счастью, этот метод тоже гарантированно возвращает сериализуемый список.)

Возвращаемый тип `Product` инкапсулирует описание, цену и место нахождение продукта, как видно в листинге 10.6.

Обратите внимание на то, что класс Product поддается сериализации. Сервер создает объект Product, а клиент получает его копию (рис. 10.7).

Листинг 10.6. Содержимое файла Product.java

```
1.  import java.io.*;
2.  public class Product implements Serializable
3.  {
4.    public Product(String description, double price)
5.    {
6.      this.description = description;
7.      this.price = price;
8.    }
9.    public String getDescription()
10.   {
11.     return description;
12.   }
13.
14.   public double getPrice()
15.   {
16.     return price;
17.   }
18.   public Warehouse getLocation()
19.   {
20.     return location;
21.   }
22.   public void setLocation(Warehouse location)
23.   {
24.     this.location = location;
25.   }
26.
27.   private String description;
28.   private double price;
29.   private Warehouse location;
30.  }
```

Однако существует одна тонкость. У класса Product имеется поле экземпляра типа Warehouse, то есть удаленного интерфейса. Объект Warehouse не поддается сериализации, а также может иметь огромное количество данных о состоянии. Поэтому вместо него клиент получает объект-заглушку, ссылающийся на удаленный объект Warehouse. Этот объект-заглушка может отличаться от объекта-заглушки centralWarehouse, на котором вызывался метод getProduct. В нашей реализации будет два вида продуктов: тостеры (toasters) и книги (books), находящиеся в разных хранилищах.

Динамическая загрузка классов

В следующем примере программы демонстрируется еще одна тонкость. Список строк с ключевыми словами отправляется серверу, в результате чего из хранилища возвращается экземпляр класса Product. Конечно, клиентской программе будет необходим файл класса Product.class для компиляции. Однако на случай невозможности отыскать подходящий по ключевым словам вариант наша серверная программа предусматривает возврат объекта такого продукта, который точно должен понравиться всем, а именно — книги *Core Java*. Этот объект является экземпляром класса Book, представляющего собой подкласс класса Product.

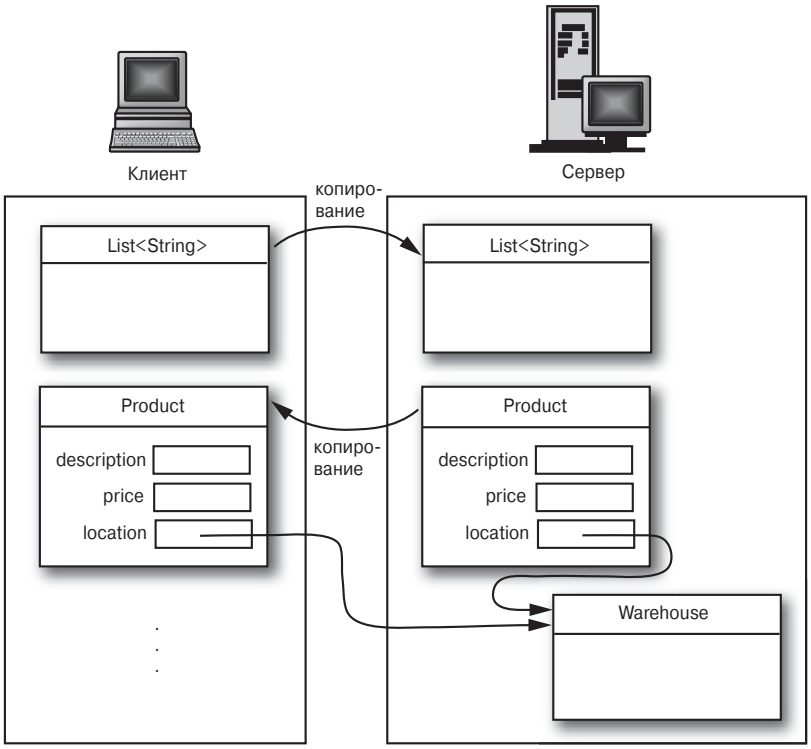


Рис. 10.7. Копирование локальных объектов параметров и результатов

На момент компиляции клиент может и не знать о существовании класса `Book`. Однако при запуске он должен быть способен выполнять методы `Book`, которые переопределяют методы `Product`. Это показывает, что клиенту необходимо иметь возможность загружать дополнительные классы во время выполнения. Для этого в клиенте применяется тот же механизм, что и в службе реестра RMI. Классы обслуживаются Web-сервером, класс сервера RMI сообщает клиенту их URL-адреса, а тот за тем на их основании делает HTTP-запросы для загрузки соответствующих файлов классов.

Всякий раз, когда программа загружает новый код из другого места в сети, возникает угроза безопасности. Из-за этого во всех приложениях RMI с динамической загрузкой классов требуется обязательно использовать *диспетчер защиты*. (Более подробно о диспетчерах защиты и загрузчиках классов рассказывалось в главе 9.)

Программы, в которых применяется RMI, должны устанавливать диспетчер защиты для осуществления контроля за действиями загружаемых динамическим образом классов. Обеспечивается такая установка с помощью следующей инструкции:

```
System.setSecurityManager(new SecurityManager());
```



Если все классы доступны локальным образом, тогда диспетчер защиты в принципе не нужен. Когда все необходимые программе файлы классов известны на этапе развертывания, они все могут развертываться локально. Однако часто бывает так, что со временем клиентская или серверная программа совершенствуется, и в нее добавляются новые классы. В таких случаях динамическая загрузка классов оказывается очень выгодным подходом. Во всех случаях, когда код загружается из другого источника, необходим диспетчер защиты.

По умолчанию SecurityManager не позволяет никакому коду из программы устанавливать сетевые соединения. Однако программе требуется устанавливать сетевые соединения с тремя следующими местами:

- с Web-сервером, который отвечает за загрузку удаленных классов;
- со службой реестра RMI;
- с удаленными объектами.

Для предоставления разрешения на выполнение этих операций создается файл политики. (О файлах политики более подробно рассказывалось в главе 9.) Ниже приведен файл политики, позволяющий приложению устанавливать любые сетевые соединения через порт с номером, как минимум, 1024. (RMI по умолчанию использует порт 1099, а удаленные объекты обычно запрашивают порты с номерами выше 1024. Для загрузки классов мы используем порт 8080.)

```
grant

{

permission java.net.SocketPermission

"*:1024-65535", "connect";

};
```

Далее необходимо указать диспетчеру защиты, что оно должен считывать файл политики путем установки свойства java.security.policy в имя этого файла. Вы можете использовать для этого такой вызов:

```
System.setProperty("java.security.policy", "rmi.policy");
```

В качестве альтернативного варианта, можете задать параметр системного свойства в командной строке:

```
-Djava.security.policy=rmi.policy
```

Чтобы запустить это демонстрационное приложение, сначала удостоверьтесь в том, что уничтожили программы реестра RMI, Web-сервера и сервера, которые создавали для предыдущего примера, а затем откройте четыре окна консоли и выполните следующие шаги.

1. Скомпилируйте файлы исходного кода для классов интерфейса, реализации, клиента и сервера:

   ```
   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. В первом окне консоли перейдите в тот каталог, в котором нет никаких файлов классов, и запустите реестр RMI.
4. Во втором окне консоли перейдите в каталог `download` и запустите `NanoHTTPD`.
5. В третьем окне консоли перейдите в каталог `server` и запустите сервер:

```
java -Djava.rmi.server.codebase=http://localhost:8080/WarehouseServer
```

6. В четвертом окне консоли перейдите в каталог `client` и запустите клиента:

```
java WarehouseClient
```

В листинге 10.7 показан код класса `Book`. Обратите внимание на то, что метод `getDescription` переопределяется так, чтобы он отображал номер ISBN. Благодаря этому, при запуске клиентская программа показывает номер ISBN для книги *Core Java*, а это подтверждает то, что класс `Book` был загружен динамически. В листинге 10.8 приведен код реализации хранилища данных (`Warehouse`). У него имеется ссылка на резервное хранилище. В случае если элемент не удается отыскать в центральном хранилище, поиск начинает выполняться в резервном хранилище. В листинге 10.9 приведен код серверной программы. Только центральное хранилище входит в реестр RMI. Обратите внимание, что удаленная ссылка на резервное хранилище может передаваться клиенту даже несмотря на то, что она и не включена в реестр RMI. Подобное происходит всякий раз, когда не удается отыскать никаких соответствий по ключевым словам и клиенту отправляется книга *Core Java* (в поле `location` которой содержится ссылка на резервное хранилище данных).

Листинг 10.7. Содержимое файла `Book.java`

```
1.  /**
2.   * Книга - это продукт с номером ISBN.
3.   * @version 1.0 2007-10-09
4.   * @author Cay Horstmann
5.   */
6.   public class Book extends Product
7.  {
8.    public Book(String title, String isbn, double price)
9.    {
10.     super(title, price);
11.     this.isbn = isbn;
12.   }
13.
14.   public String getDescription()
15.   {
16.     return super.getDescription() + " " + isbn;
17.   }
18.
19. private String isbn;
20. }
```

Листинг 10.8. Содержимое файла `WarehouseImpl.java`

```
1.  import java.rmi.*;
2.  import java.rmi.server.*;
3.  import java.util.*;
4.
5.  /**
6.   * Этот класс отвечает за реализацию удаленного интерфейса Warehouse.
7.   * @version 1.0 2007-10-09
8.   * @author Cay Horstmann
9.   */
10. public class WarehouseImpl extends UnicastRemoteObject implements Warehouse
11. {
12.   /**
13.    * Создает реализацию хранилища данных.
14.    */
15.   public WarehouseImpl(Warehouse backup) throws RemoteException
16.   {
17.     products = new HashMap<String, Product>();
18.     this.backup = backup;
19.   }
20.
21.   public void add(String keyword, Product product)
22.   {
23.     product.setLocation(this);
24.     products.put(keyword, product);
25.   }
26.
27.  public double getPrice(String description) throws RemoteException
28.  {
29.     for (Product p : products.values())
30.       if (p.getDescription().equals(description)) return p.getPrice();
31.     if (backup == null) return 0;
32.     else return backup.getPrice(description);
33.   }
34.
35.   public Product getProduct(List<String> keywords) throws RemoteException
36.   {
37.     for (String keyword : keywords)
38.     {
39.       Product p = products.get(keyword);
40.       if (p != null) return p;
41.     }
42.     if (backup != null)
43.       return backup.getProduct(keywords);
44.     else if (products.values().size() > 0)
45.       return products.values().iterator().next();
46.     else
47.       return null;
48.   }
49.
50.   private Map<String, Product> products;
51.   private Warehouse backup;
52. }
```

Листинг 10.9. Содержимое файла `WarehouseServer.java`

```
1.  import java.rmi.*;
2.  import javax.naming.*;
3.  /**
4.   * Эта серверная программа создает экземпляры удаленных объектов
5.   * Warehouse, регистрирует их с помощью службы имен и ожидает,
6.   * когда клиенты начнут вызывать методы.
7.   * @version 1.12 2007-10-09
8.   * @author Cay Horstmann
9.   */
10. public class WarehouseServer
11. {
12.   public static void main(String[] args) throws RemoteException, NamingException
13.   {
14.     System.setProperty("java.security.policy", "server.policy");
15.     System.setSecurityManager(new SecurityManager());
16.     System.out.println("Constructing server implementation...");
17.                  // Создание серверной реализации.
18.     WarehouseImpl backupWarehouse = new WarehouseImpl(null);
19.     WarehouseImpl centralWarehouse = new WarehouseImpl(backupWarehouse);
20.
21.     centralWarehouse.add("toaster", new Product("Blackwell Toaster", 23.95));
22.     backupWarehouse.add("java", new Book("Core Java vol. 2", "0132354799", 44.95));
23.
24.     System.out.println("Binding server implementation to registry...");
25.                   // Связывание серверной реализации с реестром.
26.     Context namingContext = new InitialContext();
27.     namingContext.bind("rmi:central_warehouse", centralWarehouse);
28.
29.     System.out.println("Waiting for invocations from clients...");
30.                   // Ожидание вызовов от клиентов.
31.   }
32. }
```

Удаленные ссылки с множеством интерфейсов

Удаленный класс может реализовать множество интерфейсов. Для примера возьмем удаленный интерфейс `ServiceCenter`:

```
public interface ServiceCenter extends Remote
{
  int getReturnAuthorization(Product prod) throws RemoteException;
}
```

Теперь предположим, что класс `WarehouseImpl` реализует этот интерфейс так же, как и интерфейс `Warehouse`. При передаче удаленной ссылки на такой сервисный центр другой виртуальной машине, получатель приобретает объект-заглушку, имеющий доступ к удаленным методам как в интерфейсе `ServiceCenter`, так и в интерфейсе `Warehouse`. Для выяснения того, реализует ли конкретно данный удаленный объект такой интерфейс, можно использовать операцию `instanceof`. Предположим, что удаленный объект получается посредством переменной типа `Warehouse`:

```
Warehouse location = product.getLocation();
```

Этот удаленный объект может представлять, а может и не представлять сервисный центр (`ServiceCenter`). Выяснить это можно с помощью следующего теста:

```
if (location instanceof ServiceCenter)
```

Если тест пройдет успешно, можно приводить `location` к типу `ServiceCenter` и вызывать метод `getReturnAuthorization`.

Удаленные объекты и методы `equals()`, `hashCode()` и `clone()`

Объекты, вставляемые наборами, должны обязательно переопределять метод `equals()`. В случае использования хеш-набора (hash set) или хеш-таблицы (hash map) должен еще также определяться и метод `hashCode()`. При сравнении удаленных объектов, однако, возникает одна проблема. Для выяснения того, имеют ли два удаленных объекта одинаковое содержимое, вызов метода `equals()` будет требовать установки соединения с обслуживающими эти объекты серверами и выполнения сравнивания их содержимого на предмет равенства. Как и вызов любого удаленного метода, вызов этого метода может завершиться неудачей. Но метод `equals()` в классе `Object` не предусматривает выдачи в таком случае исключения `RemoteException`, а все методы в удаленном интерфейсе обязательно должны уметь выдавать такое исключение. Из-за того, что метод подчиненного класса не может генерировать больше исключений, чем метод заменяемого им суперкласса, определять метод `equals()` в удаленном интерфейсе нельзя. То же самое касается и метода `hashCode()`.

Вместо этого методы `equals()` и `hashCode()` в объектах-заглушках должны проверять просто место размещения удаленных объектов, и если те ссылаются на один и тот же удаленный объект, считать их равными, а если на разные, то считать их не равными даже в том случае, если те имеют одинаковое содержимое. Аналогичным образом хеш-код тоже должен вычисляться только из идентификатора объекта.

По тем же техническим причинам в удаленных ссылках нельзя использовать и метод `clone()`. При необходимости делать удаленный вызов и с его помощью вынуждать сервер клонировать объект реализации, методу `clone()` необходимо иметь возможность выдавать исключение `RemoteException`. Однако в суперклассе `Object` ему не разрешено генерировать никаких исключений, кроме `CloneNotSupportedException`.

Таким образом, получается, что использовать удаленные ссылки в наборах и хеш-таблицах можно, но при этом всегда следует помнить о том, что при выполнении сравнения на предмет равенства и хеширования содержимое удаленных объектов учитываться не должно, и что клонировать удаленные ссылки просто не возможно.

Активизация удаленных объектов

В предыдущих примерах для создания экземпляров и регистрации объектов так, чтобы клиенты могли вызывать их удаленным образом, использовалась программа-сервер. Однако в некоторых случаях подход с созданием множества экземпляров удаленных объектов и ожиданием подключений к ним, независимо от того, требуются они клиентским объектам или нет, может быть неэффективным. Механизм *активизации* позволяет отсрочивать создание объектов так, чтобы удаленный объект конструировался только в случае поступления запроса на вызов в нем удаленного метода от хотя бы одного клиента.

В случае применения такого механизма клиентский код остается полностью неизменным. Клиент просто запрашивает удаленную ссылку и использует ее для вызовов.

Серверная программа, однако, заменяется программой *активизации*, создающей для объектов, которые может потребоваться создать в более позднее время, *дескрипторы активизации* и связывающей получателей удаленных вызовов с помощью службы имен. При поступлении первого вызова содержащаяся в этом дескрипторе информация применяется для создания соответствующего объекта.

Каждый удаленный объект, который используется подобным образом, должен расширять не класс UnicastRemoteObject, а класс Activatable, а также, конечно, реализовать один или более удаленных интерфейсов. Например:

```
class WarehouseImpl
  extends Activatable
  implements Warehouse
{
  . . .
}
```

Поскольку создание объекта откладывается на более позднее время, оно должно происходить стандартизированным образом. Для этого требуется предоставить конструктор, принимающий два следующих параметра:

- идентификатор активизации (который просто передается конструктору суперкласса);
- один объект, содержащий всю касающуюся процесса создания информацию и помещенный в оболочку MarshalledObject.

При необходимости использовать множество параметров создания, их все обязательно нужно упаковывать в один объект. Для этой цели всегда можно использовать массив Object[] или ArrayList.

При построении дескриптора активизации создавать MarshalledObject из информации о конструировании можно следующим образом:

```
MarshalledObject<T> param = new MarshalledObject<T>(constructionInfo);
```

Для получения десериализированной версии этой информации в конструкторе объекта реализации необходимо использовать метод get() класса MarshalledObject:

```
T constructionInfo = param.get();
```

Чтобы продемонстрировать, как работает механизм активизации, мы изменяем класс WarehouseImpl так, чтобы в роли информации о создании выступала таблица описаний и цен. Всю эту информацию мы упаковываем в оболочку MarshalledObject, а затем извлекаем ее оттуда в 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.");
                // Реализация хранилища создана.
}
```

Путем передачи в конструкторе суперкласса в качестве второго параметра значения 0 мы указываем, что библиотека RMI должна назначать прослушивающему порту подходящий номер.

Этот конструктор предусматривает отображение на экране сообщения, позволяющего видеть, что объект хранилища был действительно активизирован по запросу.



На самом деле обеспечивать, чтобы удаленные объекты расширяли класс Activatable, вовсе необязательно. Вместо этого можно и просто помещать в конструктор класса сервера вызов статического метода:

```
Activatable.exportObject(this, id, 0);
```

Теперь давайте перейдем к программе активизации. Первым делом потребуется определить группу активизации и описать в ней общие параметры для запуска виртуальной машины, содержащей удаленные объекты. Наиболее важным из этих параметров является политика обеспечения безопасности.

Создать дескриптора группы активизации можно следующим образом:

```
Properties props = new Properties();
props.put("java.security.policy", "/path/to/server.policy");
ActivationGroupDesc group = new ActivationGroupDesc(props, null);
```

С помощью второго параметра описываются специальные командные опции. В нашем примере таковые не нужны, поэтому мы и передаем просто ссылку `null`.

Далее необходимо создать идентификатор группы с помощью следующего вызова:

```
ActivationGroupID id = ActivationGroup.getSystem().registerGroup(group);
```

После этого можно приступать непосредственно к созданию самих дескрипторов активизации. Для этого для каждого объекта, который должен создаваться по запросу, нужно предоставить следующие сведения:

- идентификатор группы активизации для виртуальной машины, в которой должен создавать данный объект;
- имя класса (например, `WarehouseImpl"` или `"com.mycompany.MyClassImpl"`);
- строку с URL-адресом, по которому должны загружаться файлы классов (это должен быть базовый URL-адрес, не включающий никаких путей к пакетам);
- маршализированную информацию для конструктора.

Например:

```
MarshalledObject param = new MarshalledObject(constructionInfo);
ActivationDesc desc = new ActivationDesc(id, "WarehouseImpl",
                      "http://myserver.com/download/", param);
```

Далее этот дескриптор необходимо передать статическому методу `Activatable.register()`. Он возвратит объект какого-то класса, который реализует удаленные интерфейсы класса реализации, после чего его можно будет связать с именем с помощью службы имен, как показано ниже:

```
Warehouse centralWarehouse = (Warehouse) Activatable.register(desc);
namingContext.bind("rmi:central_warehouse", centralWarehouse);
```

В отличие от приводившихся в предыдущих примерах серверных программ, работа программы активизации завершается после регистрации и связывания получателей, а создание всех удаленных объектов происходит только при возникновении первого вызова удаленного метода.

В листингах 10.10 и 10.11 показан код программы активизации и реализации хранилища данных с возможностью активизации. Код интерфейса хранилища данных и клиентской программы остается без изменений.

Чтобы запустить эту демонстрационную программу, потребуется выполнить следующие шаги.

1. Скомпилировать все исходные файлы.
2. Распределить файлы классов 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. Запустить службу реестра RMI из каталога `rmi` (в котором не содержится никаких файлов классов).
4. Запустить демон активизации RMI опять-таки из каталога `rmi`:

   ```
   rmid -J-Djava.security.policy=rmid.policy
   ```

   Программа `rmid` перехватывает запросы на активизацию объектов и затем активизирует соответствующие объекты в отдельной виртуальной машине. Для запуска виртуальной машины программе `rmid` необходимы определенные полномочия; эти полномочия задаются в файле политики (листинг 10.12). Параметр `-J` используется для передачи параметров виртуальной машине, под управлением которой был запущен демон активизации.
5. Запустить Web-сервер `NanoHTTPD` из каталога `download`.
6. Запустить программу активизации из каталога `server`:

   ```
   java -Djava.rmi.server.codebase=http://localhost:8080/WarehouseActivator
   ```

   Эта программа завершит свою работу после регистрации и связывания получателей запросов на активизацию с помощью службы имен. (Вас наверняка интересует, зачем нужно указывать кодовую базу, если она и так уже указывается в конструкторе дескриптора активизации. Дело в том, что та информация обрабатывается только демоном активизации RMI. Поэтому она все равно необходима службе регистрации RMI для загрузки классов удаленных интерфейсов.)
7. Запустить клиентскую программу из каталога `client`:

   ```
   java WarehouseClient
   ```

   Она отобразит на экране уже знакомое описание продуктов. При первом запуске этой программы в окне оболочки демона активизации можно также будет увидеть и все сообщения конструктора.

Листинг 10.10. Содержимое файла `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 создает экземпляр удаленного объекта
9.   * Warehouse регистрирует его с помощью службы имен и ожидает,
10.  * когда клиенты начнут вызывать методы.
11.  * @version 1.12 2007-10-09
12.  * @author Cay Horstmann
13.  */
14.
```

```
15. public class WarehouseActivator
16. {
17.   public static void main(String[] args) throws RemoteException, NamingException,
18.     ActivationException, IOException
19.   {
20.     System.out.println("Constructing activation descriptors...");
21.                   // Создание дескрипторов активизации.
22.     Properties props = new Properties();
23.                   // Использование файла server.policy из текущего каталога.
24.     props.put("java.security.policy", new  File("server.policy").getCanonicalPath());
25.     ActivationGroupDesc group = new ActivationGroupDesc(props, null);
26.     ActivationGroupID id = ActivationGroup.getSystem().registerGroup(group);
27.     Map<String, Double> prices = new HashMap<String, Double>();
28.     prices.put("Blackwell Toaster", 24.95);
29.     prices.put("ZapXpress Microwave Oven", 49.95);
30.
31.     MarshalledObject<Map<String, Double>> param = new
32.           MarshalledObject<Map<String, Double>>(prices);
33.
34.     String codebase = "http://localhost:8080/";
35.
36.     ActivationDesc desc = new ActivationDesc(id, "WarehouseImpl", codebase, param);
37.     Warehouse centralWarehouse = (Warehouse) Activatable.register(desc);
38.
39.     System.out.println("Binding activable implementation to registry...");
40.                   // Связывание допускающей активизацию реализации с реестром.
41.     Context namingContext = new InitialContext();
42.     namingContext.bind("rmi:central_warehouse", centralWarehouse);
43.     System.out.println("Exiting...");
44.                   // Завершение работы.
45.   }"
46. }
```

Листинг 10.11. Содержимое файла `WarehouseImpl.java`

```
1.  import java.io.*;
2.  import java.rmi.*;
3.  import java.rmi.activation.*;
4.  import java.util.*;
5.
6.  /**
7.   * Этот класс отвечает за реализацию удаленного
8.   * интерфейса хранилища (Warehouse).
9.   * @version 1.0 2007-10-20
10.  * @author Cay Horstmann
11. */
12. public class WarehouseImpl extends Activatable implements Warehouse
13. {
14.   public WarehouseImpl(ActivationID id, MarshalledObject<Map<String, Double>> param)
15.     throws RemoteException, ClassNotFoundException, IOException
16.   {
17.     super(id, 0);
18.     prices = param.get();
19.     System.out.println("Warehouse implementation constructed.");
20.                   // Реализация хранилища создана.
21.   }
22.
```

```
23.   public double getPrice(String description) throws RemoteException
24.   {
25.     Double price = prices.get(description);
26.     return price == null ? 0 : price;
27.   }
28.
29.   private Map<String, Double> prices;
30. }
```

Листинг 10.12. Содержимое файла `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.  }
```

API java.rmi.activation.Activatable 1.2

- `protected Activatable(ActivationID id, int port)`

  Создает активизируемый объект и устанавливает обработчик, связанный с указанным портом. Если для параметра порта задается значение `0`, порт назначается автоматически.

- `static Remote exportObject(Remote obj, ActivationID id, int port)`

  Превращает удаленный объект в активизируемый. Возвращает объект, доступный для удаленных обращений. Для того чтобы порт был назначен автоматически, нужно задать для его номера значение `0`.

- `static Remote register(ActivationDesc desc)`

  Регистрирует дескриптор для активизируемого объекта и готовит его для получения удаленных вызовов. Возвращает объект, который будет доступен для удаленных обращений.

API java.rmi.MarshalledObject 1.2

- `MarshalledObject(Object obj)`

  Создает объект, содержащий сериализованные данные указанного объекта.

- `Object get()`

  Десериализует хранящиеся данные и возвращает объект.

API 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 опции командной строки. Если специальные установки не нужны, используется значение `null`.

API **java.rmi.activation.ActivationGroup 1.2**

- `static ActivationSystem getSystem()`

  Возвращает ссылку на систему активизации.

API **java.rmi.activation.ActivationSystem 1.2**

- `ActivationGroupID registerGroup(ActivationGroupDesc group)`

  Регистрирует группу активизации и возвращает идентификатор группы.

API **java.rmi.activation.ActivationDesc 1.2**

- `ActivationDesc(ActivationGroupID id, String className, String classFileURL, MarshalledObject data)`

  Создает дескриптор активизации.

Web-службы и JAX-WS

За последние годы популярной технологией для вызовов удаленных методов стали *Web-службы* (Web services). С технической точки зрения каждая Web-служба состоит из двух компонентов:

- непосредственно самой службы, доступ к которой может получаться с помощью транспортного протокола SOAP;
- описания службы в формате WSDL.

SOAP — это XML-протокол для вызова удаленных методов, подобный тому, который применяется в RMI для связи между клиентами и серверами. Как приложения RMI можно программировать, не зная ничего о деталях протокола RMI, так и Web-службы можно вызывать, не зная ничего о деталях протокола SOAP.

WSDL — это язык описания интерфейсов. Он тоже основан на XML. В WSDL-документе описывается интерфейс Web-службы: методы, которые могут вызываться, а также их параметры и возвращаемые типы. В этом разделе мы сгенерируем документ WSDL из службы, реализованной на языке Java. В этом документе будет содержаться вся информация, которая может быть необходима клиентской программе для вызова данной службы, независимо от того, написана ли она на Java или на каком-то другом языке программирования. А в следующем разделе мы создадим Java-программу, вызывающую службу электронной коммерции Amazon, с помощью предоставляемого компанией Amazon документа WSDL. То, на каком языке реализована данная служба, нам совершенно не известно.

Использование JAX-WS

В Java доступно несколько наборов инструментальных средств для реализации Web-служб. В этом разделе мы рассмотрим технологию JAX-WS, которая входит в состав Java SE 6 и выше.

В случае применения JAX-WS интерфейс для Web-службы не предоставляется. Вместо этого класс снабжается аннотацией `@WebService`, как показано в листинге 10.13. Помимо этого, параметр `description` снабжается аннотацией `@WebParam`. Благодаря этому данный параметр приобретает в файле WSDL понятный для человека вид. (Эта аннотация является не обязательной. По умолчанию данный параметр назывался бы `arg0`.)

Листинг 10.13. Содержимое файла `Warehouse.java`

```
1.  package com.horstmann.corejava;
2.  import java.util.*;
3.  import javax.jws.*;
4.  /**
5.   * Этот класс отвечает за реализацию Web-службы Warehouse.
6.   * @version 1.0 2007-10-09
7.   * @author Cay Horstmann
8.  */
9.
10. @WebService
11. public class Warehouse
12. {
13.   public Warehouse()
14.   {
15.     prices = new HashMap<String, Double>();
16.     prices.put("Blackwell Toaster", 24.95);
17.     prices.put("ZapXpress Microwave Oven", 49.95);
18.   }
19.
20.   public double getPrice(@WebParam(name="description") String description)
21.   {
22.     Double price = prices.get(description);
23.     return price == null ? 0 : price;
24.   }
25.
26.   private Map<String, Double> prices;
27. }
```

В RMI классы заглушек генерировались динамическим образом, но в случае JAX-WS для их генерации запускается специальная утилита, а именно — `wsgen` с предварительным переходом в базовый каталог источника `Webservices1`, как показано ниже:

```
wsgen -classpath . com.horstmann.corejava.Warehouse
```



Утилита `wsgen` требует, чтобы класс, который предоставляет Web-службу, содержался в пакете, отличном от пакета по умолчанию.

Эта утилита генерирует два довольно обычных класса в пакете `com.horstmann.corejava.jaxws`. Первый класс инкапсулирует все параметры вызова:

```
public class GetPrice
{
  private String description;
  public String getDescription() { return this.description; }
  public void setDescription(String description) { this.description = description; }
}
```

Что касается второго класса, то он инкапсулирует возвращаемое значение:

```
public class GetPriceResponse
{
  private double _return;
  public double get_return() { return this._return; }
  public void set_return(double _return) { this._return = _return; }
}
```

Обычно для развертывания Web-служб применяется какая-то сложная серверная инфраструктура, но мы таковую здесь рассматривать не будем. В JDK имеется очень простой механизм для тестирования Web-служб. Все, что он требует — это просто вызывать метод `Endpoint.publish`. Это приводит к запуску сервера по указанному URL-адресу (см. листинг 10.14).

Листинг 10.14. Содержимое файла `WarehouseServer.java`

```
1.  package com.horstmann.corejava;
2.
3.  import javax.xml.ws.*;
4.
5.  public class WarehouseServer
6.  {
7.    public static void main(String[] args)
8.    {
9.      Endpoint.publish("http://localhost:8080/WebServices/warehouse",
10.                     new Warehouse());
11.   }
12. }
```

На этом этапе требуется скомпилировать серверные классы, выполнить `wsgen` и запустить сервер:

```
java com.horstmann.corejava.WarehouseServer
```

Далее следует указать Web-браузеру адрес `http://localhost:8080/WebServices/warehouse?wsdl`. Это приведет к получению следующего файла WSDL:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:tns="http://corejava.horstmann.com/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             targetNamespace="http://corejava.horstmann.com/"
             name="WarehouseService">
  <types>
   <xsd:schema>
    <xsd:import schemaLocation="http://localhost:8080/WebServices/warehouse?xsd=1"
                namespace="http://corejava.horstmann.com/"></xsd:import>
   </xsd:schema>
  </types>
  <message name="getPrice">
   <part element="tns:getPrice" name="parameters"></part>
  </message>
  <message name="getPriceResponse">
   <part element="tns:getPriceResponse" name="parameters"></part>
  </message>
  <portType name="Warehouse">
    <operation name="getPrice">
    <input message="tns:getPrice"></input>
    <output message="tns:getPriceResponse"></output>
   </operation>
  </portType>
  <binding name="WarehousePortBinding" type="tns:Warehouse">
   <soap:binding style="document"
                 transport="http://schemas.xmlsoap.org/soap/http"></soap:binding>
   <operation name="getPrice">
```

```
    <soap:operation soapAction=""></soap:operation>
    <input><soap:body use="literal"></soap:body></input>
     <output><soap:body use="literal"></soap:body></output>
    </operation>
  </binding>
  <service name="WarehouseService">
   <port name="WarehousePort" binding="tns:WarehousePortBinding">
    <soap:address location="http://localhost:8080/WebServices/warehouse"></soap:address>
   </port>
  </service>
</definitions>
```

Данное описание показывает, что предоставляется операция `getPrice`. Ее входные данные выглядят как `tns:getPrice`, а выходные — как `tns:getPriceResponse`. (Здесь `tns` — это псевдоним для целевого пространства имен `http://corejava.horstmann.com`.)

Чтобы разобраться в этих типах, нужно ввести в окне браузера следующий адрес: `http://localhost:8080/WebServices/warehouse?xsd=1`. Это приведет к отображению такого документа XSL:

```
<xs:schema targetNamespace="http://corejava.horstmann.com/" version="1.0">
 <xs:element name="getPrice" type="tns:getPrice"/>
 <xs:element name="getPriceResponse" type="tns:getPriceResponse"/>
 <xs:complexType name="getPrice">
  <xs:sequence><xs:element name="description"
                           type="xs:string" minOccurs="0"/></xs:sequence>
 </xs:complexType>
 <xs:complexType name="getPriceResponse">
  <xs:sequence><xs:element name="return" type="xs:double"/></xs:sequence>
 </xs:complexType>
</xs:schema>
```

Здесь видно, что у `getPrice` есть элемент `description` типа `string`, а у `getPriceResponse` — элемент `return` типа `double`.



Что именно делает служба, в файле WSDL не указывается. В нем указываются только типы параметров и возвращаемых значений.

Клиент Web-службы

Теперь давайте обратимся к реализации клиента. Главное помнить о том, что клиенту о сервере не известно ничего, кроме того, что содержится в файле WSDL. Для создания классов Java, способных взаимодействовать с сервером, с помощью утилиты `wsimport` генерируется набор клиентских классов.

```
wsimport -keep -p com.horstmann.corejava.server
   http://localhost:8080/WebServices/warehouse?wsdl
```

Опция `-keep` указывает сохранить исходные файлы на случай, если их понадобится просмотреть. Генерируемые классы и интерфейсы выглядят следующим образом:

```
GetPrice
GetPriceResponse
Warehouse
WarehouseService
ObjectFactory
```

Классы `GetPrice` и `GetPriceResponse` вы уже видели.

Интерфейс Warehouse определяет удаленный метод getPrice:

```
public interface Warehouse
{
  @WebMethod public double getPrice(@WebParam(name = "description") String description);
}
```

О классе WarehouseService необходимо знать только одну вещь: его метод getPort выдает объект-заглушку Warehouse, посредством которого и вызывается служба (см. листинг 10.15).

На класс ObjectFactory, а также файл package-info.java, определяющий аннотацию уровня пакета, пока можно не обращать внимание. (Об аннотациях более подробно будет рассказываться в главе 11.)



Для сгенерированных классов можно использовать любой удобный пакет. Если вы присмотритесь повнимательнее, то заметите, что классы GetPrice и GetPriceResponse находятся в разных пакетах на сервере и клиенте. Проблемой это не является. А все потому, что ни серверу, ни клиенту ничего не известно о Java-реализациях друг друга. Им даже не известно то, на Java ли они реализованы.

Листинг 10.15. Содержимое файла WarehouseClient.java

```
1.  import java.rmi.*;
2.  import javax.naming.*;
3.  import com.horstmann.corejava.server.*;
4.
5.  /**
6.   * Клиент для программы Warehouse.
7.   * @version 1.0 2007-10-09
8.   * @author Cay Horstmann
9.   */
10. public class WarehouseClient
11. {
12.   public static void main(String[] args) throws NamingException, RemoteException
13.   {
14.     WarehouseService service = new WarehouseService();
15.     Warehouse port = service.getPort(Warehouse.class);
16.
17.     String descr = "Blackwell Toaster";
18.     double price = port.getPrice(descr);
19.     System.out.println(descr + ": " + price);
20.   }
21. }
```

Теперь можно уже и запускать клиентскую программу. Для этого нужно дважды удостовериться в том, что сервер по-прежнему работает, открыть еще одно окно оболочки и выполнить следующую команду:

```
java WarehouseClient
```

Это приведет к отображению уже знакомого сообщения о цене тостера.



Тем, у кого возник вопрос о том, почему здесь нет никакого эквивалента реестру RMI, объясняем. При обнаружении места расположения удаленного объекта для RMI, клиенту не требуется знать о том, на каком именно сервере находится объект. Ему нужно знать только то, как найти реестр. Для вызова Web-службы, однако, клиенту необходимо знать URL-адрес сервера. Поэтому он и жестко кодируется в классе WarehouseService.

Для того чтобы иметь возможность увидеть, как на самом деле происходит взаимодействие между клиентом и сервером, мы использовали сетевой анализатор пакетов (рис. 10.8).

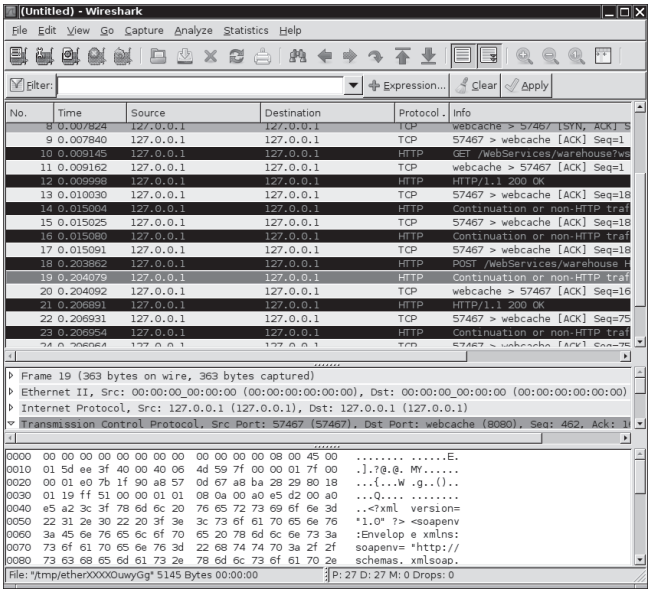


Рис. 10.8. Анализ трафика SOAP

Клиент отправляет серверу следующий запрос:

```
<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:ns1="http://corejava.horstmann.com/">
 <soapenv:Body>
  <ns1:getPrice><description>Blackwell Toaster</description></ns1:getPrice>
 </soapenv:Body>
</soapenv:Envelope>
```

Сервер ему отвечает следующим образом:

```
<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:ns1="http://corejava.horstmann.com/">
 <soapenv:Body>
  <ns1:getPriceResponse><return>24.95</return></ns1:getPriceResponse>
 </soapenv:Body>
</soapenv:Envelope>
```

В общем, в этом разделе были рассмотрены такие основные характеристики Web-служб.

- Эти службы определяются в документе WSDL, который форматируется как документ XML.
- В методах самих запросов и ответов используется SOAP — еще один формат XML.
- Программы клиентов и серверов могут писаться на любом языке.

Служба электронного магазина Amazon

Чтобы сделать обсуждение Web-служб более интересным, давайте рассмотрим конкретный пример, а именно – Web-службу электронной коммерции Amazon, полное описание которой можно найти по адресу `http://www.amazon.com/gp/aws/landing.html`. Данная служба позволяет программистам реализовать взаимодействие с системой Amazon и решать разнообразные задачи. Например, вы можете получить список всех книг с указанием автора и названия, заказать книгу и т.д. Данная служба доступна компаниям, которые обслуживают заказчиков, используя систему Amazon в качестве исполняющих средств. Для запуска примера, рассматриваемого в этом разделе, вы должны зарегистрироваться на Amazon и получить идентификатор независимого разработчика. Это позволит обращаться к Web-службе.

При необходимости вы можете использовать описанный здесь подход для взаимодействия с любой другой Web-службой. Список, включающий многие свободно доступные службы, которые можно применять для проверки программы, доступен по адресу `http://www.xmethods.com`.

Теперь давайте более детально рассмотрим WSDL-файл службы Amazon (найти который можно по адресу `http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl`). Описание операции `ItemSearch` в нем выглядит так:

```
<operation name="ItemSearch">
  <input message="tns:ItemSearchRequestMsg"/>
  <output message="tns:ItemSearchResponseMsg"/>
</operation>
...
<message name="ItemSearchRequestMsg">
  <part name="body" element="tns:ItemSearch"/>
</message>
<message name="ItemSearchResponseMsg">
  <part name="body" element="tns:ItemSearchResponse"/>
</message>
```

А определения типов `ItemSearch` и `ItemSearchResponse` в нем выглядят следующим образом:

```
<xs:element name="ItemSearch">
  <xs:complexType>
   <xs:sequence>
     <xs:element name="MarketplaceDomain" type="xs:string" minOccurs="0"/>
     <xs:element name="AWSAccessKeyId" type="xs:string" minOccurs="0"/>
     <xs:element name="SubscriptionId" type="xs:string" minOccurs="0"/>
     <xs:element name="AssociateTag" type="xs:string" minOccurs="0"/>
     <xs:element name="XMLEscaping" type="xs:string" minOccurs="0"/>
     <xs:element name="Validate" type="xs:string" minOccurs="0"/>
     <xs:element name="Shared" type="tns:ItemSearchRequest" minOccurs="0"/>
     <xs:element name="Request" type="tns:ItemSearchRequest" minOccurs="0"
                 maxOcurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="ItemSearchResponse">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="tns:OperationRequest" minOccurs="0"/>
      <xs:element ref="tns:Items" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

С помощью технологии JAX-WS операция `ItemSearch` превращается в вызов метода:

```
void itemSearch(String marketPlaceDomain, String awsAccessKeyId,
  String subscriptionId, String associateTag, String xmlEscaping, String validate,
  ItemSearchRequest shared, List<ItemSearchRequest> request,
  Holder<OperationRequest> opHolder, Holder<List<Items>> responseHolder)
```

Тип параметра `ItemSearchRequest` определяется следующим образом:

```
<xs:complexType name="ItemSearchRequest">
  <xs:sequence>
   <xs:element name="Actor" type="xs:string" minOccurs="0"/>
    <xs:element name="Artist" type="xs:string" minOccurs="0"/>
    . . .
    <xs:element name="Author" type="xs:string" minOccurs="0"/>
    . . .
   <xs:element name="ResponseGroup" type="xs:string" minOccurs="0"
               maxOccurs="unbounded"/>
    . . .
    <xs:element name="SearchIndex" type="xs:string" minOccurs="0"/>
    . . .
</xs:complexType>
```

Это описание преобразуется в показанный ниже класс:

```
public class ItemSearchRequest
{
  public ItemSearchRequest() { ... }
  public String getActor() { ... }
  public void setActor(String newValue) { ... }
  public String getArtist() { ... }
  public void setArtist(String newValue) { ... }
  ...
  public String getAuthor() { ... }
  public void setAuthor(String newValue) { ... }
  ...
  public List<String> getResponseGroup() { ... }
  ...
  public void setSearchIndex(String newValue) { ... }
  ...
}
```

Для вызова поисковой службы создается объект `ItemSearchRequest` и вызывается метод `itemSearch` объекта порта:

```
ItemSearchRequest request = new ItemSearchRequest();
request.getResponseGroup().add("ItemAttributes");
request.setSearchIndex("Books");
Holder<List<Items>> responseHolder = new Holder<List<Items>>();
request.setAuthor(name);
port.itemSearch("", accessKey, "", "", "", "", request, null, null, responseHolder);
```

Объект порта преобразует объект Java в сообщение SOAP, передает его серверу Amazon, затем преобразует возвращенное сообщение в объект `ItemSearchResponse` и помещает ответ в объект-"заполнитель".



Предлагаемая Amazon документация по параметрам и возвращаемым значениям является чрезвычайно краткой. Тем не менее, увидеть, как выглядят запросы и ответы SOAP, можно и просто путем заполнения форм по адресу `http://awszone.com/scratchpads/index.aws`. Они помогают понять, какие значения параметров нужно предоставлять, и какие возвращаемые значения можно ожидать.

Наше демонстрационное приложение (код которого приведен в листинге 10.16) выглядит довольно просто. Пользователь указывает имя автора и щелкает на кнопке Search (Найти). В результате он получает первую страницу ответа (рис. 10.9). Это свидетельствует о том, что Web-служба функционирует и взаимодействие с ней происходит нормально. При желании вы можете самостоятельно расширить возможности этого приложения.



Рис. 10.9. Подключение к Web-службе

Чтобы запустить эту программу, вам придется сначала сгенерировать искусственные клиентские классы:

```
wsimport -p com.horstmann.amazon
http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl
```

После этого необходимо отредактировать файл `AmazonTest.java` так, чтобы он включал полученный вами от Amazon ключ, скомпилировать его и запустить:

```
javac AmazonTest.java
java AmazonTest
```

Листинг 10.16. Содержимое файла `AmazonTest.java`

```
1.  import com.horstmann.amazon.*;
2.  import java.awt.*;
3.  import java.awt.event.*;
4.  import java.util.List;
5.  import javax.swing.*;
6.  import javax.xml.ws.*;
7.
8.  /**
9.   * Клиент для тестовой программы электронной коммерции Amazon.
10.  * @version 1.10 2007-10-20
11.  * @author Cay Horstmann
12.  */
13. public class AmazonTest
14. {
15.   public static void main(String[] args)
16.   {
17.     JFrame frame = new AmazonTestFrame();
18.     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.     frame.setVisible(true);
20.   }
21. }
```

```
22. /**
23.  * Фрейм для выбора автора книги и отображения ответа сервера.
24.  */
25. class AmazonTestFrame extends JFrame
26. {
27.  public AmazonTestFrame()
28.  {
29.    setTitle("AmazonTest");
30.    setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
31.    JPanel panel = new JPanel();
32.    panel.add(new JLabel("Author:"));
33.    author = new JTextField(20);
34.    panel.add(author);
35.
36.    JButton searchButton = new JButton("Search");
37.    panel.add(searchButton);
38.    searchButton.addActionListener(new ActionListener()
39.      {
40.        public void actionPerformed(ActionEvent event)
41.        {
42.          result.setText("Please wait...");
43.          new SwingWorker<Void, Void>()
44.            {
45.              @Override
46.              protected Void doInBackground() throws Exception
47.            {
48.              String name = author.getText();
49.              String books = searchByAuthor(name);
50.              result.setText(books);
51.              return null;
52.            }
53.         }.execute();
54.      }
55.    });
56.  result = new JTextArea();
57.  result.setLineWrap(true);
58.  result.setEditable(false);
59.  if (accessKey.equals("your key here"))
60.                // Здесь должен быть ваш ключ.
61.  {
62.    result.setText("You need to edit the Amazon access key.");
63.                // Необходимо отредактировать ключ доступа Amazon.
64.    searchButton.setEnabled(false);
65.  }
66.  add(panel, BorderLayout.NORTH);
67.  add(new JScrollPane(result), BorderLayout.CENTER);
68. }
69. /**
70.  * Вызов Web-службы Amazon для поиска соответствующих автору заголовков книг.
71.  * @param name Имя автора
72.  * @return Описание заголовков, которые должны считаться соответствующими
73.  */
74.  private String searchByAuthor(String name)
75.  {
76.    AWSECommerceService service = new AWSECommerceService();
77.    AWSECommerceServicePortType port =
78.        service.getPort(AWSECommerceServicePortType.class);
```

```
79.     ItemSearchRequest request = new ItemSearchRequest();
80.     request.getResponseGroup().add("ItemAttributes");
81.     request.setSearchIndex("Books");
82.
83.    Holder<List<Items>> responseHolder = new Holder<List<Items>>();
84.     request.setAuthor(name);
85.     port.itemSearch("", accessKey, "", "", "", "", request, null, null,
                        responseHolder);
86.
87.     List<Item> response = responseHolder.value.get(0).getItem();
88.
89.     StringBuilder r = new StringBuilder();
90.     for (Item item : response)
91.     {
92.       r.append("authors=");
93.       List<String> authors = item.getItemAttributes().getAuthor();
94.       r.append(authors);
95.       r.append(",title=");
96.       r.append(item.getItemAttributes().getTitle());
97.       r.append(",publisher=");
98.       r.append(item.getItemAttributes().getPublisher());
99.       r.append(",pubdate=");
100.      r.append(item.getItemAttributes().getPublicationDate());
101.      r.append("\n");
102.    }
103.    return r.toString();
104.  }
105.
106.  private static final int DEFAULT_WIDTH = 450;
107.  private static final int DEFAULT_HEIGHT = 300;
108.
109.  private static final String accessKey = "12Y1EEATQ8DDYJCVQYR2";
110.
111.  private JTextField author;
112.  private JTextArea result;
113. }
```

Этот пример показывает, что вызов Web-службы, по сути, выполняется точно так же, как и вызов любого удаленного метода. Программист вызывает локальный метод на прокси-объекте, а тот затем устанавливает соединение сервером. Поскольку Web-службы появляются повсюду, это свидетельствует о том, что они, несомненно, являются интересной технологией для разработчиков приложений.

В этой главе был рассмотрен механизм RMI, представляющий собой сложную модель программирования для приложений Java, которая широко применяется в архитектуре Java EE, а также механизм Web-служб, которые позволяют подключаться к клиентам и серверам не зависящим от языка программирования образом. В следующей главе речь пойдет о другом аспекте Java-программирования, а именно — о возможности взаимодействия на той же самой машине с "собственным" (платформенно-ориентированным) кодом, написанном на другом языке программирования.

Глава 11

НАПИСАНИЕ СЦЕНАРИЕВ, КОМПИЛЯЦИЯ КОДА И ОБРАБОТКА АННОТАЦИЙ

В ЭТОЙ ГЛАВЕ...

- Написание сценариев для платформы Java
- API-интерфейс компилятора
- Использование аннотаций
- Синтаксис аннотаций
- Стандартные аннотации
- Обработка аннотаций на уровне исходного кода
- Разработка байт-кода

В этой главе рассматриваются три методики для обработки кода. API-интерфейс сценариев позволяет вызывать код на языке сценариев вроде JavaScript или Groovy, API-интерфейс компилятора дает возможность компилировать код Java внутри приложения, а обработчики аннотаций — обрабатывать содержащие аннотации файлы классов или исходного кода на Java. Здесь можно будет увидеть, что существует очень много приложений для обработки аннотаций, начиная от простой диагностики и заканчивая "разработкой байт-кода", вставкой байтовых кодов в файлы классов и даже выполнением программ.

Написание сценариев для платформы Java

Язык сценариев — это такой язык, который позволяет избегать обычного цикла операций редактирования, компиляции, связывания и выполнения за счет осуществления интерпретации текста программы во время выполнения. Языки сценариев обладают рядом преимуществ.

- Быстрый цикл работы, стимулирующий желание к экспериментированию.
- Возможность изменения поведения выполняющейся программы.
- Возможность выполнения настройки пользователями программы.

С другой стороны, у большинства языков программирования отсутствуют функции, необходимые для программирования сложных приложений, вроде строгого контроля типов, инкапсуляции и модульности.

Из-за этого существует соблазн объединять преимущества языков сценариев с преимуществами традиционных языков. Именно это и позволяет делать API-интерфейс сценариев (scripting API) платформы Java. В частности, он предоставляет возможность вызывать из программы Java сценарии, написанные на JavaScript, Groovy, Ruby и даже таких экзотических языках, как Scheme и Haskell. (Наличие возможности работы в обратном направлении, то есть возможности получения доступа к Java из языка сценариев, зависит уже от поставщика конкретно данного языка сценариев. У большинства языков сценариев, способных выполняться на виртуальной машине Java, такая возможность есть.)

В следующих разделах мы покажем, как можно выбирать механизм для конкретного языка, как можно выполнять сценарии и как можно пользоваться дополнительными преимуществами, которые предлагают некоторые механизмы сценариев.

Получение механизма сценариев

Механизм сценариев (scripting engine) — это, по сути, библиотека, которая позволяет выполнять сценарии на определенном языке. При запуске виртуальная машина обнаруживает все доступные механизмы сценариев. Получить их список можно путем создания `ScriptEngineManager` и вызова метода `getEngineFactories`. Далее у каждой фабрики механизмов можно запрашивать информацию об именах поддерживаемых механизмов, типах MIME и расширениях файлов. В табл. 11.1 показаны наиболее типичные значения.

Обычно необходимый механизм известен и тогда он может запрашиваться просто по имени, типу MIME или расширению, например:

```
ScriptEngine engine = manager.getEngineByName("JavaScript");
```

Таблица 11.1. Свойства фабрик механизмов сценариев

Механизм	Имена	Типы MIME	Расширения
Rhino (входит в состав Java SE 6)	js, rhino, JavaScript, javascript, ECMAScript, ecmascript	application/javascript, application/ecmascript, text/javascript, text/ecmascript	js
Groovy	groovy	Отсутствуют	groovy
SISC Scheme	scheme, sisc	Отсутствуют	scc, sce, scm, shp

В составе Java SE 6 предлагается механизм Rhino, представляющий собой разработанный корпорацией Mozilla интерпретатор JavaScript. Добавлять дополнительные языки тоже можно путем указания в пути к классам необходимых JAR-файлов. Обычно будет требоваться два набора JAR-файлов: один для реализации самого языка сценариев, а второй — для реализации механизма, адаптирующего этот язык под API сценариев. На сайте `http://scripting.dev.java.net` предлагаются механизмы для огромного количества самых разных языков сценариев. Чтобы добавить поддержку для Groovy, например, потребуется просто указать в пути к классам *`groovy`*`/lib/*` (с сайта `http://groovy.codehaus.org`) и `groovy-engine.jar` (с сайта `http://scripting.dev.java.net`).

API `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)`

  Получает механизм сценариев с заданным именем, расширением файла сценария или типом MIME.

API `javax.script.ScriptEngineFactory 6`

- `List<String> getNames()`
- `List<String> getExtensions()`
- `List<String> getMimeTypes()`

  Получает имена, расширения файлов сценариев и типы MIME, которые ассоциируются с данной фабрикой.

Оценка сценариев и привязки

Получив механизм, можно приступать к вызову сценария, что делается следующим образом:

```
Object result = engine.eval(scriptString);
```

Если сценарий хранится в файле, тогда нужно открыть `Reader` и вызвать

```
Object result = engine.eval(reader);
```

С помощью одного и того же механизма можно вызывать множество сценариев. Если какой-то из сценариев содержит определения переменных, функций или классов, большинство механизмов сценариев будет сохранять их для последующего использования. Например

```
engine.eval("n = 1728");
Object result = engine.eval("n + 1");
```

вернет 1729.



Узнавать, является ли параллельное выполнение сценариев во множестве потоков безопасным, можно путем вызова

```
Object param = factory.getParameter("THREADING");
```

Возвращаться будет одно из перечисленных ниже значений.

- `null`: параллельное выполнение является небезопасным.
- `"MULTITHREADED"`: параллельное выполнение является безопасным. Результаты одного потока могут быть видны из другого потока.
- `"THREAD-ISOLATED"`: то же самое, что и `"MULTITHREADED"`, но только для каждого потока поддерживаются отдельные привязки переменных.
- `"STATELESS"`: то же самое, что и `"THREAD-ISOLATED"`, но только сценарии не могут изменять привязки переменных.

К механизму часто требуется добавлять привязки переменных. Каждая привязка состоит из имени и ассоциированного объекта Java. Например, возьмем следующие операторы:

```
engine.put(k, 1728);
Object result = engine.eval("k + 1");
```

Код сценария будет считывать определение `k` из привязок в "области действия механизма" (engine scope). Это очень важно, поскольку 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("f.text = 'Ok'");
```

И, наоборот, извлекать переменные, которые были привязаны операторами сценария, тоже можно:

```
engine.eval("n = 1728");
Object result = engine.get("n");
```

Помимо области действия механизма, еще также существует и глобальная область действия. Любые привязки, которые добавляются к `ScriptEngineManager`, становятся видимыми для всех механизмов.

Вместо того чтобы добавлять привязки в глобальную область действия или область действия механизма, их еще также можно собирать в объекте типа `Bindings` и передавать методу `eval`:

```
Bindings scope = engine.createBindings();
scope.put(b, new JButton());
engine.eval(scriptString, scope);
```

Такой подход является очень удобным, если сохранять набор привязок для будущих вызовов метода `eval` не требуется.



Конечно, может возникнуть необходимость иметь и другие области действия, отличные от глобальной области действия и области действия механизма. Например, Web-контейнеру могут быть нужны области действия запросов и сеансов. В таких случаях разработчик должен все делать сам, а именно — реализовать класс, реализующий интерфейс `ScriptContext` и управлять набором своих областей действия. Каждая область действия должна снабжаться целочисленным номером, и поиск в первую очередь должен выполняться в тех из них, которые имеют более низкий номер. (В стандартной библиотеке доступен только класс `SimpleScriptContext`, но он предусматривает лишь глобальную область действия и область действия механизма.)

API javax.script.ScriptEngine 6

- `Object eval(String script)`
- `Object eval(Reader reader)`
- `Object eval(String script, Bindings bindings)`
- `Object eval(Reader reader, Bindings bindings)`

  Оценивает сценарий, предоставляемый в виде строки или механизмом считывания с учетом заданных привязок.
- `Object get(String key)`
- `void put(String key, Object value)`

  Получает или помещает привязку в область действия механизма.
- `Bindings createBindings()`

  Создает пустой объект `Bindings`, подходящий для данного механизма.

API javax.script.ScriptEngineManager 6

- `Object get(String key)`
- `void put(String key, Object value)`

  Получает или помещает привязку в глобальную область действия.

API javax.script.Bindings 6

- `Object get(String key)`
- `void put(String key, Object value)`

  Получает или помещает привязку в область действия, представляемую данным объектом `Bindings`.

Перенаправление входных и выходных данных

Стандартные входные и выходные данные сценария могут перенаправляться путем вызова в контексте сценария метода `setReader` и `SetWriter`, например:

```
StringWriter writer = new StringWriter();
engine.getContext().setWriter(new PrintWriter(writer, true));
```

Любые выходные данные, записываемые с помощью таких функций JavaScript, как `print` или `println`, отправляются объекту `writer`.



Методу `setWriter` можно передавать любой объект `Writer`, но если это не `PrintWriter`, механизм Rhino будет выдавать исключение.

Методы `setReader` и `setWriter` воздействуют только на стандартные источники входных и выходных данных механизма сценариев. Например, в случае выполнения следующего кода JavaScript

```
println("Hello");
java.lang.System.out.println("World");
```

перенаправлены будут только первые выходные данные.

Механизм Rhino не имеет понятия о стандартном источнике входных данных. Вызов `setReader` не будет иметь никакого эффекта.

API javax.script.ScriptEngine 6

- `ScriptContext getContext()`

  Получает стандартный контекст сценариев для данного механизма.

API javax.script.ScriptContext 6

- `Reader getReader()`
- `void setReader(Reader reader)`
- `Writer getWriter()`
- `void setWriter(Writer writer)`
- `Writer getErrorWriter()`
- `void setErrorWriter(Writer writer)`

  Получает или устанавливает средство считывания для входных данных или средство записи для обычных или информирующих об ошибке выходных данных.

Вызов сценарных функций и методов

В случае многих механизмов сценариев функция может вызываться на языке сценариев и без оценки фактического кода сценария. Такой подход является удобным, если пользователям разрешается реализовывать службу на том языке сценариев, на котором они предпочитают сами.

Те механизмы сценариев, которые предлагают такую функциональную возможность, реализуют интерфейс `Invocable`. В частности, механизм Rhino реализует этот интерфейс.

Чтобы вызвать функцию, нужно просто вызвать метод `invokeFunction` и указать в нем имя и параметры желаемой функции:

```
if (engine implements Invocable)
  ((Invocable) engine).invokeFunction("aFunction", param1, param2);
```

Если язык сценариев является объектно-ориентированным, этот метод можно вызвать следующим образом:

```
((Invocable) engine).invokeMethod(implicitParam, "aMethod", explicitParam1,
  explicitParam2);
```

Здесь объект `implicitParam` представляет собой прокси-объект соответствующего объекта в языке сценариев. Это может быть результат предшествующего вызова механизма сценариев.



Если механизм сценариев не реализует интерфейс `Invocable`, вызывать метод не зависящим от языка образом все равно может быть возможно. Метод `getMethodCallSyntax` класса `ScriptEngineFactory` генерирует строку, которую можно передавать методу `eval`. Однако все параметры этого методы должны быть обязательно привязаны к именам, в то время как `invokeMethod` может вызываться с произвольными значениями.

Можно, конечно, пойти на один шаг дальше и попросить механизм сценариев реализовать интерфейс Java. Тогда станет возможно вызывать сценарные функции и методы с помощью Java-синтаксиса вызова методов.

Детали зависят от механизма сценария, но обычно необходимо просто предоставлять функцию для каждого метода интерфейса. Например, возьмем следующий интерфейс Java:

```
public interface Greeter
{
  String greet(String whom);
}
```

Предположим, что в Rhino для него предоставляется следующая функция

```
function greet(x) { return "Hello, " + x + "!"; }
```

Этот код должен оцениваться первым. Далее может вызываться

```
Greeter g = ((Invocable) engine).getInterface(Greeter.class);
```

После этого может осуществляться простой вызов метода Java

```
String result = g.greet("World");
```

В результате незаметно будет вызываться JavaScript-метод `greet`. Этот подход похож на выполнение вызова удаленного метода, о котором более подробно рассказывалось в главе 10.

В объектно-ориентированном языке программирования доступ к классу сценария может получаться посредством соответствующего интерфейса Java. Например, возьмем следующий код JavaScript, определяющий класс `SimpleGreeter`:

```
function SimpleGreeter(salutation) { this.salutation = salutation; }
SimpleGreeter.prototype.greet =
   function(whom) { return this.salutation + ", " + whom + "!"; }
```

Этот класс можно использовать для создания приветственных сценариев с разными приветствиями (вроде `Hello`, `Goodbye` и т.д.).



Более подробно о способах определения классов на JavaScript можно узнать в книге Дэвида Флэнагана (David Flanagan) под названием *JavaScript — The Definitive Guide, 5th Edition* (O'Reilly, 2006 г.).

После оценивания определения класса JavaScript можно вызвать

```
Object goodbyeGreeter = engine.eval("new SimpleGreeter('Goodbye')");
Greeter g = ((Invocable) engine).getInterface(goodbyeGreeter, Greeter.class);
```

При вызове `g.greet("World")` на JavaScript-объекте `goodbyeGreeter` будет вызываться метод `greet`, в результате чего будет возвращаться строка `"Goodbye, World!"`.

Таким образом, получается, что интерфейс `Invocable` является очень полезным в тех случаях, когда код сценария требуется вызывать из Java, не заботясь о синтаксисе языке сценариев.

API javax.script.Invocable 6

- `Object invokeFunction(String name, Object... parameters)`
- `Object invokeMethod(Object implicitParameter, String name, Object... explicitParameters)`

  Вызывает функцию или метод с заданным именем, передавая указанные параметры.

- `<T> T getInterface(Class<T> iface)`

  Возвращает реализацию заданного интерфейса, реализуя методы с помощью функций в механизме сценариев.

- `<T> T getInterface(Object implicitParameter, Class<T> iface)`

  Возвращает реализацию заданного интерфейса, реализуя методы с помощью методов заданного объекта.

Компиляция сценария

Некоторые механизмы сценариев умеют компилировать код сценария в промежуточную форму для более эффективного выполнения. Такие механизмы реализуют интерфейс `Compilable`. Следующий пример демонстрирует, как можно компилировать и оценивать код, содержащийся 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. Следующий код демонстрирует, как выполнять скомпилированный сценарий в случае, если компиляция прошла успешно, и исходный сценарий, если оказалось, что механизм не поддерживает компиляцию.

```
if (script != null)
  script.eval();
else
  engine.eval(reader);
```

Конечно, компилировать сценарий требуется только в том случае, если необходимо, чтобы он выполнялся повторно.

API javax.script.Compilable 6

- `CompiledScript compile(String script)`
- `CompiledScript compile(Reader reader)`

  Компилирует сценарий, представленный строкой или устройством считывания.

API javax.script.CompiledScript 6

- `Object eval()`
- `Object eval(Bindings bindings)`

  Оценивает данный сценарий.

Пример: создание сценария обработки событий GUI-интерфейса

Чтобы проиллюстрировать API-интерфейс сценариев, мы разработаем демонстрационную программу, позволяющую пользователям задавать обработчики событий на выбираемом (предпочитаемом) ими языке сценариев.

Взгляните на код, приведенный в листинге 11.1. Здесь класс `ButtonFrame` похож на демонстрировавшуюся в первом томе программу обработки событий, но имеет два следующих отличия.

- У каждого компонента имеется свой собственный набор свойств `name`.
- Не содержит никаких обработчиков событий.

Необходимые обработчики определяются в файле свойств. Определение каждого свойства имеет следующий вид:

```
имяКомпонента.имяСобытия = кодСценария
```

Например, в случае выбора для использования языка JavaScript, обработчики событий предоставляются в файле `js.properties` следующим образом:

```
yellowButton.action=panel.background = java.awt.Color.YELLOW
blueButton.action=panel.background = java.awt.Color.BLUE
redButton.action=panel.background = java.awt.Color.RED
```

В сопроводительном коде также имеются файлы и для Groovy и SISC Scheme.

Начинается программа с загрузки механизма для языка, который указывается в командной строке. Если никакой язык не указывается, используется JavaScript.

Далее выполняется обработка сценария `init.`*`язык`*, если таковой присутствует. В общем, это кажется вполне хорошей идеей. Более того, интерпретатор Scheme нуждается в кое-каких громоздких инициализациях, которые нам вовсе не хотелось включать в каждый сценарий обработки событий.

После этого осуществляется рекурсивный обход по всем дочерним компонентам и добавление привязок (*`имя, объект`*) в область действия механизма.

Далее выполняется считывание файла *`язык`*`.properties`. Для каждого свойства синтезируется прокси-объект обработчика событий, который, собственно говоря, и заставляет код сценария выполняться. Детали носят немного технический характер, поэтому тем, кто желает разобраться во всем на техническом уровне, рекомендуем еще раз прочитать посвященный прокси-объектам раздел в главе 6 первого тома и посвященный событиям JavaBeans раздел в главе 8 этого тома. Главная суть, однако, состоит в том, что каждый обработчик событий вызывает

```
engine.eval(scriptCode);
```

Давайте более подробно остановимся на `yellowButton`. При обработке строки

```
yellowButton.action=panel.background = java.awt.Color.YELLOW
```

у нас появляется компонент `JButton` с именем `"yellowButton"`. Мы далее берем его и присоединяем к нему объект `ActionListener` с методом `actionPerformed`, который выполняет сценарий

```
panel.background = java.awt.Color.YELLOW
```

Механизм содержит привязку, которая связывает имя `"panel"` с объектом `JPanel`. Когда возникает событие, выполняется метод `setBackground` этого объекта-панели и цвет изменяется.

Запустить эту программу с использованием обработчиков событий JavaScript можно просто путем выполнения следующей команды:

```
java ScriptTest
```

Чтобы использовать обработчики событий Groovy, потребуется выполнить следующую команду:

```
java -classpath .:groovy/lib/*:jsr223-engines/groovy/build/groovy-engine.jar
ScriptTest groovy
```

Здесь *groovy* — это каталог, в который был установлен Groovy, а *jsr223-engines* — каталог, в котором находятся адаптеры механизма с сайта `http://scripting.dev.java.net`.

Чтобы испробовать Scheme, нужно будет загрузить SISC Scheme с сайта `http://sisc-scheme.org/` и выполнить такую команду:

```
java -classpath .:sisc/*:jsr223-engines/scheme/build/scheme-engine.jar
ScriptTest scheme
```

Данное приложение демонстрирует, как можно использовать сценарии при программировании GUI-интерфейсов на платформе Java. Желающие могут пойти на один шаг дальше и описать GUI с помощью файла XML, как было показано в главе 2. Тогда эта программа превратиться в интерпретатор для GUI-интерфейсов с визуальным представлением, определяемым XML, и поведением, определяемым языком сценариев. Нельзя не обратить внимания на сходство со средой динамических серверных сценариев и динамических HTML-страниц.

Листинг 11.1. Содержимое файла `ScriptTest.java`

```
1.  import java.awt.*;
2.  import java.beans.*;
3.  import java.io.*;
4.  import java.lang.reflect.*;
5.  import java.util.*;
6.  import javax.script.*;
7.  import javax.swing.*;
8.  /**
9.   * @version 1.00 2007-10-28
10.  * @author Cay Horstmann
11.  */
12. public class ScriptTest
13. {
14.  public static void main(final String[] args)
15.  {
16.   EventQueue.invokeLater(new Runnable()
17.   {
18.     public void run()
19.     {
20.       String language;
21.       if (args.length == 0) language = "js";
22.       else language = args[0];
23.
24.       ScriptEngineManager manager = new ScriptEngineManager();
25.       System.out.println("Available factories: ");
26.       for (ScriptEngineFactory factory : manager.getEngineFactories())
27.         System.out.println(factory.getEngineName());
28.       final ScriptEngine engine = manager.getEngineByName(language);
29.       if (engine == null)
30.       {
31.         System.err.println("No engine for " + language);
32.         System.exit(1);
33.       }
```

```
34.       ButtonFrame frame = new ButtonFrame();
35.
36.       try
37.       {
38.         File initFile = new File("init." + language);
39.         if (initFile.exists())
40.         {
41.           engine.eval(new FileReader(initFile));
42.         }
43.
44.         getComponentBindings(frame, engine);
45.
46.         final Properties events = new Properties();
47.         events.load(new FileReader(language + ".properties"));
48.         for (final Object e : events.keySet())
49.         {
50.           String[] s = ((String) e).split("\\.");
51.           addListener(s[0], s[1], (String) events.get(e), engine);
52.         }
53.       }
54.       catch (Exception e)
55.       {
56.         e.printStackTrace();
57.       }
58.
59.       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
60.       frame.setTitle("ScriptTest");
61.       frame.setVisible(true);
62.     }
63.   });
64. }
65.
66. /**
67.  * Собирает все именованные компоненты в контейнер.
68.  * @param c the component
69.  * @param namedComponents
70.  */
71. private static void getComponentBindings(Component c, ScriptEngine engine)
72. {
73.   String name = c.getName();
74.   if (name != null) engine.put(name, c);
75.   if (c instanceof Container)
76.   {
77.     for (Component child : ((Container) c).getComponents())
78.       getComponentBindings(child, engine);
79.   }
80. }
81. /**
82.  * Добавляет слушателя к объекту, метод слушателя
83.  * которого выполняет сценарий.
84.  * @param beanName Имя bean-компонента, к которому
85.  * должен быть добавлен слушатель.
86.  * @param eventName Наименование типа слушателя,
87.  * вроде "action" (действие) или "change" (изменение).
88.  * @param scriptCode Подлежащий выполнению код сценария.
89.  * @param engine Механизм, который должен отвечать за выполнение кода.
90.  * @param bindings Привязки для выполнения.
91.  */
```

```
92.   private static void addListener(String beanName, String eventName,
93.   final String scriptCode, final ScriptEngine engine)
94.     throws IllegalArgumentException, IntrospectionException,
95.     IllegalAccessException, InvocationTargetException
96.   {
97.     Object bean = engine.get(beanName);
98.     EventSetDescriptor descriptor = getEventSetDescriptor(bean, eventName);
99.     if (descriptor == null) return;
100.    descriptor.getAddListenerMethod().invoke(
101.      bean,
102.      Proxy.newProxyInstance(null, new Class[] { descriptor.getListenerType() },
103.      new InvocationHandler()
104.      {
105.        public Object invoke(Object proxy, Method method, Object[] args)
106.             throws Throwable
107.        {
108.          engine.eval(scriptCode);
109.          return null;
110.        }
111.      }));
112.  }
113.  private static EventSetDescriptor getEventSetDescriptor(Object bean, String eventName)
114.       throws IntrospectionException
115.  {
116.    for (EventSetDescriptor descriptor : Introspector.getBeanInfo(bean.getClass())
117.         .getEventSetDescriptors())
118.      if (descriptor.getName().equals(eventName)) return descriptor;
119.    return null;
120.  }
121. }
122. class ButtonFrame extends JFrame
123. {
124.   public ButtonFrame()
125.   {
126.     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
127.     panel = new JPanel();
128.     panel.setName("panel");
129.     add(panel);
130.
131.     yellowButton = new JButton("Yellow");
132.     yellowButton.setName("yellowButton");
133.     blueButton = new JButton("Blue");
134.     blueButton.setName("blueButton");
135.     redButton = new JButton("Red");
136.     redButton.setName("redButton");
137.
138.     panel.add(yellowButton);
139.     panel.add(blueButton);
140.     panel.add(redButton);
141.   }
142.   public static final int DEFAULT_WIDTH = 300;
143.   public static final int DEFAULT_HEIGHT = 200;
144.   private JPanel panel;
145.   private JButton yellowButton;
146.   private JButton blueButton;
147.   private JButton redButton;
148. }
```

API-интерфейс компилятора

В предыдущих разделах мы показали, как можно взаимодействовать с кодом на языке сценариев. Теперь мы предлагаем рассмотреть другой сценарий, а именно — программы Java, компилирующие Java-код. Инструментальных средств, которым необходимо вызывать компилятор Java, существует довольно много, например, к их числу относятся:

- среды разработки;
- обучающие и тренировочные программы по Java;
- автоматизированные средства разработки и тестирования;
- средства создания шаблонов, обрабатывающие фрагменты Java-кода, например, страницы JSP (JavaServer Pages).

Раньше приложения вызывали компилятор Java за счет обращения к недокументированным классам в библиотеке *jdk*/lib/tools.jar. Начиная с версии Java SE 6, общедоступный API-интерфейс для компиляции является частью платформы Java, и использовать tools.jar больше не требуется. Об этом API-интерфейсе компилятора как раз и пойдет речь в этом разделе.

Компилирование простым путем

Вызывается компилятор очень просто, например:

```
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
OutputStream outStream = ..., errStream = ...;
int result = compiler.run(null, outStream, errStream, "-sourcepath", "src", "Test.java");
```

Результирующее значение 0 указывает на то, что компиляция прошла успешно.

Все выходные данные и сообщения об ошибках компилятор отправляет заданным потокам. Для этих параметров может устанавливаться и значение null, в случае чего используются потоки System.out и System.err. Первый параметр метода run представляет поток входных данных. Поскольку никаких входных данных с консоли компилятор не принимает, для этого параметра всегда оставляется значение null. (Метод run наследуется от обобщенного интерфейса Tool, который допускает использование средств, считывающих входные данные.)

Остальные параметры метода run являются просто аргументами, которые нужно было бы передавать javac, если бы этот метод вызывался в командной строке. Они могут представлять собой как опции, так и имена файлов.

Использование заданий на компиляцию

С помощью объекта CompilationTask можно иметь даже еще больший контроль над процессом компиляции. В частности, с его помощью можно:

- управлять источником программного кода, например, путем предоставления кода не в файле, а в конструкторе строк (string builder);
- управлять размещением файлов классов, например, путем их сохранения в базе данных;
- обеспечивать прослушивание на предмет появления во время компиляции предупреждений и сообщений об ошибках;
- запускать компилятор в фоновом режиме.

За размещение исходных файлов и файлов классов отвечает JavaFileManager. В его обязанности входит определение для этих файлов экземпляров JavaFileObject. Каждый объект JavaFileObject может как соответствовать какому-нибудь определенному файлу на диске, так и предоставлять какой-то другой механизм для считывания и записи его содержимого.

Для обеспечения прослушивания на предмет появления сообщений об ошибках устанавливается DiagnosticListener. Всякий раз, когда компилятор уведомляет о появлении предупреждения или сообщения об ошибке, этот слушатель получает объект Diagnostic. Реализует такой интерфейс класс DiagnosticCollector. Он просто собирает все диагностические данные, чтобы их можно было просмотреть после завершения процесса компиляции.

Объект Diagnostic содержит информацию о местонахождении проблемы (в виде имени файла, номера строки и номера столбца), а также понятное для человеческого глаза описание.

Получается объект CompilationTask путем вызова метода getTask класса JavaCompiler и требует указания следующих аргументов:

- объекта Writer для любых выходных данных компилятора, не являющихся диагностическими (Diagnostic), или значения null для использования System.err;
- объекта JavaFileManager или значения null для использования стандартного диспетчера файлов компилятора;
- объекта DiagnosticListener;
- строк опций или значения null, если никакие опции не нужны;
- имен классов для обработки аннотаций или значения null, если таковых нет (об обработке аннотаций более подробно будет рассказываться чуть позже в этой главе);
- экземпляров JavaFileObject для исходных файлов.

Последние три аргумента необходимо предоставлять в виде объектов Iterable. Например, последовательность опций можно указать следующим образом:

```
Iterable<String> options = Arrays.asList("-g", "-d", "classes");
```

В качестве альтернативного варианта можно использовать любой класс коллекции.

Если требуется, чтобы компилятор считывал исходные файлы с диска, тогда можно попросить StandardJavaFileManager преобразовывать строки с именами файлов или объекты File в экземпляры JavaFileObject, например:

```
StandardJavaFileManager fileManager =
  compiler.getStandardFileManager(null, null, null);
Iterable<JavaFileObject> fileObjects =
  fileManager.getJavaFileObjectsFromStrings(fileNames);
```

Однако если требуется, чтобы компилятор считывал исходный код из какого-то другого места, а не из файла на диске, тогда нужно предоставлять свой собственный подкласс JavaFileObject. В листинге 11.2 показан код для объекта исходного файла с данными, содержащимися в StringBuilder. Данный класс расширяет простой класс SimpleJavaFileObject и переопределяет метод getCharContent, чтобы возвращать содержимое конструктора строк. Мы используем этот класс в демонстрационной программе, где динамически генерируем код для класса Java и затем компилируем его.

Класс CompilationTask реализует интерфейс Callable<Boolean>. Его можно как передавать классу Executor для выполнения в другом потоке, так и просто вы-

зывать с помощью метода call. Возвращаемое значение Boolean.FALSE обозначает неудачу.

```
Callable<Boolean> task = new JavaCompiler.CompilationTask(null, fileManager,
  diagnostics, options, null, fileObjects);
if (!task.call())
  System.out.println("Compilation failed");
            // Компиляция не удалась.
```

Если необходимо просто, чтобы компилятор создавал файлы классов на диске, выполнять настройку JavaFileManager не нужно. Наше демонстрационное приложение, однако, будет генерировать файлы классов в виде байтовых массивов и затем считывать их из памяти с помощью специального загрузчика классов. В листинге 11.3 показан код класса, реализующего интерфейс JavaFileObject. Его метод openOutputStream возвращает ByteArrayOutputStream, в который компилятор будет помещать байт-код.

Указать диспетчеру файлов компилятора использовать эти объекты файлов не так-то просто. Все дело в том, что в стандартной библиотеке нет класса, который бы реализовал интерфейс StandardJavaFileManager. Поэтому требуется создавать подкласс класса ForwardingJavaFileManager, делегирующий все вызовы определенному диспетчеру файлов. В нашей ситуации необходимо всего лишь изменить метод getJavaFileForOutput, и мы достигаем этого 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
    {
      Вернуть специальный объект файла
    }
  };
```

Таким образом, получается, что метод run задания JavaCompiler удобно вызывать только в тех случаях, когда требуется просто активизировать компилятор обычным образом и выполнять чтение и запись файлов на диске. Выходные данные и сообщения об ошибках можно перехватывать, но выполнять их синтаксический анализ нужно самостоятельно.

При желании иметь больше контроля над процессами обработки файлов и уведомления об ошибках, лучше вместо этого использовать класс CompilationTask. Его API-интерфейс является довольно сложным, но зато позволяет управлять каждым аспектом процесса компиляции.

Листинг 11.2. Содержимое файла StringBuilderJavaSource.java

```
1.  import java.net.*;
2.  import javax.tools.*;
3.
4.  /**
5.   * Источник Java, хранящий код в конструкторе строк.
6.   * @version 1.00 2007-11-02
7.   * @author Cay Horstmann
8.   */
9.  public class StringBuilderJavaSource extends SimpleJavaFileObject
10. {
11.   /**
```

```
12.    * Создает новый объект StringBuilderJavaSource
13.    * @param name Имя исходного файла, представляемого данным объектом файла
14.    */
15.   public StringBuilderJavaSource(String name)
16.   {
17.     super(URI.create("string:///" + name.replace('.', '/') +
18.       Kind.SOURCE.extension), Kind.SOURCE);
19.     code = new StringBuilder();
20.   }
21.
22.   public CharSequence getCharContent(boolean ignoreEncodingErrors)
23.   {
24.     return code;
25.   }
26.
27.   public void append(String str)
28.   {
29.     code.append(str);
30.     code.append('\n');
31.   }
32.   private StringBuilder code;
33. }
```

Листинг 11.3. Содержимое файла `ByteArrayJavaClass.java`

```
1.  import java.io.*;
2.  import java.net.*;
3.  import javax.tools.*;
4.  /**
5.   * Класс Java, размещающий байт-код в массиве байтов.
6.   * @version 1.00 2007-11-02
7.   * @author Cay Horstmann
8.   */
9.  public class ByteArrayJavaClass extends SimpleJavaFileObject
10. {
11.   /**
12.    * Создает новый объект ByteArrayJavaClass
13.    * @param name Имя класса, представляемого данным объектом файла
14.    */
15.   public ByteArrayJavaClass(String name)
16.   {
17.     super(URI.create("bytes:///" + name), Kind.CLASS);
18.     stream = new ByteArrayOutputStream();
19.   }
20.
21.   public OutputStream openOutputStream() throws IOException
22.   {
23.     return stream;
24.   }
25.
26.   public byte[] getBytes()
27.   {
28.     return stream.toByteArray();
29.   }
30.
31.   private ByteArrayOutputStream stream;
32. }
```

API javax.tools.Tool 6

- `int run(InputStream in, OutputStream out, OutputStream err, String... arguments)`

  Запускает утилиту с заданными входными и выходными данными, а также потоками сообщений об ошибках и аргументами. В случае успешного выполнения возвращает `0`, а в случае неудачного — ненулевое значение.

API javax.tools.JavaCompiler 6

- `StandardJavaFileManager getStandardFileManager(DiagnosticListener< ? super JavaFileObject> diagnosticListener, Locale locale, Charset charset)`

  Получает стандартный диспетчер файлов для данного компилятора. Использовать стандартные сообщения об ошибках, региональные установки и набор символов можно путем указания для соответствующих параметров 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, которое при вызове будет компилировать указанные исходные файлы. Более подробно об этом рассказывалось в предыдущем разделе.

API javax.tools.StandardJavaFileManager 6

- `Iterable<? extends JavaFileObject> getJavaFileObjectsFromStrings(Iterable<String> fileNames)`
- `Iterable<? extends JavaFileObject> getJavaFileObjectsFromFiles(Iterable<? extends File> files)`

  Преобразует последовательность имен файлов или файлов в последовательность экземпляров `JavaFileObject`.

API javax.tools.JavaCompiler.CompilationTask 6

- `Boolean call()`

  Выполняет задание на компиляцию.

API javax.tools.DiagnosticCollector<S> 6

- `DiagnosticCollector()`

  Создает пустой коллектор.

- `List<Diagnostic<? extends S>> getDiagnostics()`

  Получает собранные диагностические данные.

API javax.tools.Diagnostic<S> 6

- `S getSource()`

  Получает исходный объект, ассоциируемый с данной процедурой диагностики.

- `Diagnostic.Kind getKind()`

  Получает тип данной процедуры диагностики, которым может быть `ERROR`, `WARNING`, `MANDATORY_WARNING`, `NOTE` или `OTHER`.

- `String getMessage(Locale locale)`

  Получает сообщение, описывающее проблему, которая подлежит анализу в данной процедуре диагностики. Использовать стандартные региональные установки можно путем передачи для соответствующего параметра значения `null`.

- `long getLineNumber()`
- `long getColumnNumber()`

  Получает позицию проблемы, которая подлежит анализу в данной процедуре диагностики.

API javax.tools.SimpleJavaFileObject 6

- `CharSequence getCharContent(boolean ignoreEncodingErrors)`

  Переопределяет данный метод для объекта файла, представляющего исходный файл, и генерирует исходный код.

- `OutputStream openOutputStream()`

  Переопределяет данный метод для объекта файла, представляющего файл класса, и генерирует поток, в который могут записываться байт-коды.

API javax.tools.ForwardingJavaFileManager< M extends JavaFileManager> 6

- `protected ForwardingJavaFileManager(M fileManager)`

  Создает объект `JavaFileManager`, который делегирует все вызовы заданному диспетчеру файлов.

- `FileObject getFileForOutput(JavaFileManager.Location location, String className, JavaFileObject.Kind kind, FileObject sibling)`

  Позволяет перехватывать данный вызов при желании заменить объект файла для записи файлов классов. Параметр `kind` может принимать одно из следующих значений: `SOURCE`, `CLASS`, `HTML` или `OTHER`.

Пример: Динамическая генерация кода Java

В технологии JSP для динамических Web-страниц допускается смешивание HTML с фрагментами Java-кода, например такое:

```
<p>The current date and time is <b><%= new java.util.Date() %></b>.</p>
```

Механизм JSP динамически компилирует Java-код в сервлет. В нашем демонстрационном приложении мы используем более простой пример, и вместо этого будем генерировать динамический код Swing. Идея состоит в использовании конструктора GUI для размещения компонентов во фрейме и указания поведения компонентов во

внешнем файле. В листинге 11.4 приведен очень простой пример класса фрейма, а в листинге 11.5 — код для действий кнопок. Обратите внимание на то, что конструктор класса фрейма вызывает абстрактный метод `addEventHandlers`. Наш генератор кода генерирует подкласс, реализующий метод `addEventHandlers`, добавляя слушателя действия для каждой строки в классе `action.properties`. (Расширение генерации кода до других типов событий мы решили оставить читателю в качестве упражнения для самостоятельной работы.)

Мы помещаем этот подкласс в пакет с именем `x`, которое, надеемся, больше нигде в программе не используется. Сгенерированный код выглядит следующим образом:

```
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 для обработчика событий 1
       } } );
    // повтор для других обработчиков событий...
  } }
```

Метод `buildSource` в программе, код которой приведен в листинге 11.6, создает этот код и помещает его в объект `StringBuilderJavaSource`. Этот объект затем передается компилятору Java.

Мы используем `ForwardingJavaFileManager` с методом `getJavaFileForOutput`, который конструирует объект `ByteArrayJavaClass` для каждого класса в пакете `x`. Эти объекты перехватывают файлы классов, генерируемые при компиляции класса `x.Frame`. Данный метод добавляет каждый объект файла в список прежде, чем возвращать его, чтобы мы могли обнаружить байт-коды позже. Обратите внимание на то, что компиляция класса `x.Frame` приводит к генерации файла класса для главного класса, а также одного файла класса для каждого класса слушателя.

После компиляции мы создаем таблицу, сопоставляющую имена классов с массивами байт-кодов. Загружать хранящиеся в этой таблице классы позволяет простой загрузчик классов (код которого приведен в листинге 11.7).

Мы просим этот загрузчик загрузить класс, который только что скомпилировали, а затем создаем и отображаем класс фрейма приложения:

```
ClassLoader loader = new MapClassLoader(byteCodeMap);
Class<?> cl = loader.loadClass("x.Frame");
Frame frame = (JFrame) cl.newInstance();
frame.setVisible(true);
```

При выполнении щелчка на кнопках цвет фона меняется обычным образом. Чтобы увидеть, что действия действительно компилируются динамическим образом, измените какую-нибудь из строк в `action.properties`, например, следующим образом:

```
yellowButton=panel.setBackground(java.awt.Color.YELLOW);
yellowButton.setEnabled(false);
```

После этого запустите программу снова. Теперь кнопка `Yellow` после выполнения на ней щелчка станет недоступной. Также взгляните на каталоги кода. Вы не обнаружите там ни исходных файлов, ни файлов классов для классов из пакета `x`. Этот пример демонстрирует то, как динамическую компиляцию можно применять с сохранением файлов исходного кода и классов в памяти.

Листинг 11.4. Содержимое файла `ButtonFrame.java`

```
1.  package com.horstmann.corejava;
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 ButtonFrame()
11.   {
12.     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
13.
14.     panel = new JPanel();
15.     add(panel);
16.
17.     yellowButton = new JButton("Yellow");
18.     blueButton = new JButton("Blue");
19.     redButton = new JButton("Red");
20.
21.     panel.add(yellowButton);
22.     panel.add(blueButton);
23.     panel.add(redButton);
24.
25.     addEventHandlers();
26.   }
27.
28.   protected abstract void addEventHandlers();
29.
30.   public static final int DEFAULT_WIDTH = 300;
31.   public static final int DEFAULT_HEIGHT = 200;
32.   protected JPanel panel;
33.   protected JButton yellowButton;
34.   protected JButton blueButton;
35.   protected JButton redButton;
36. }
```

Листинг 11.5. Содержимое файла `action.properties`

```
1.  yellowButton=panel.setBackground(java.awt.Color.YELLOW);
2.  blueButton=panel.setBackground(java.awt.Color.BLUE);
3.  redButton=panel.setBackground(java.awt.Color.RED);
```

Листинг 11.6. Содержимое файла `CompilerTest.java`

```
1.  import java.awt.*;
2.  import java.io.*;
3.  import java.util.*;
4.  import java.util.List;
5.  import javax.swing.*;
6.  import javax.tools.*;
7.  import javax.tools.JavaFileObject.*;
8.
9.  /**
10.  * @version 1.00 2007-11-02
```

```
11.   * @author Cay Horstmann
12.   */
13.  public class CompilerTest
14.  {
15.    public static void main(final String[] args) throws IOException
16.    {
17.      JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
18.
19.      final List<ByteArrayJavaClass> classFileObjects =
20.        new ArrayList<ByteArrayJavaClass>();
21.
22.      DiagnosticCollector<JavaFileObject> diagnostics =
23.        new DiagnosticCollector<JavaFileObject>()
24.
25.      JavaFileManager fileManager =
26.        compiler.getStandardFileManager(diagnostics, null, null);
27.      fileManager = new ForwardingJavaFileManager<JavaFileManager>(fileManager)
28.      {
29.        public JavaFileObject getJavaFileForOutput(Location location,
30.          final String className, Kind kind, FileObject sibling) throws IOException
31.      {
32.        if (className.startsWith("x."))
33.        {
34.          ByteArrayJavaClass fileObject = new ByteArrayJavaClass(className);
35.          classFileObjects.add(fileObject);
36.          return fileObject;
37.       }
38.       else return super.getJavaFileForOutput(location, className, kind, sibling);
39.      }
40.    };
41.
42.    JavaFileObject source = buildSource("com.horstmann.corejava.ButtonFrame");
43.    JavaCompiler.CompilationTask task = compiler.getTask(null, fileManager,
44.      diagnostics, null, null, Arrays.asList(source));
45.    Boolean result = task.call();
46.
47.    for (Diagnostic<? extends JavaFileObject> d : diagnostics.getDiagnostics())
48.      System.out.println(d.getKind() + ": " + d.getMessage(null));
49.    fileManager.close();
50.    if (!result)
51.    {
52.      System.out.println("Compilation failed.");
53.                  // ("Компиляция не удалась");
54.      System.exit(1);
55.      }
56.
57.      EventQueue.invokeLater(new Runnable()
58.      {
59.        public void run()
60.        {
61.          try
62.          {
63.            Map<String, byte[]> byteCodeMap = new HashMap<String, byte[]>();
64.            for (ByteArrayJavaClass cl : classFileObjects)
65.               byteCodeMap.put(cl.getName().substring(1), cl.getBytes());
66.            ClassLoader loader = new MapClassLoader(byteCodeMap);
67.            Class<?> cl = loader.loadClass("x.Frame");
```

```
68.          JFrame frame = (JFrame) cl.newInstance();
69.          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
70.          frame.setTitle("CompilerTest");
71.          frame.setVisible(true);
72.        }
73.        catch (Exception ex)
74.        {
75.          ex.printStackTrace();
76.        }
77.      }
78.    });
79.  }
80.
81.   /**
82.    * Создает исходный код для подкласса, который
83.    * реализует метод addEventHandlers.
84.    * @return Объект файла, содержащий исходный код
85.    * в конструкторе строк.
86.    */
87.  static JavaFileObject buildSource(String superclassName) throws IOException
88.  {
89.    StringBuilderJavaSource source = new StringBuilderJavaSource("x.Frame");
90.    source.append("package x;\n");
91.    source.append("public class Frame extends " + superclassName + " {");
92.    source.append("protected void addEventHandlers() {");
93.    Properties props = new Properties();
94.    props.load(new FileReader("action.properties"));
95.    for (Map.Entry<Object, Object> e : props.entrySet())
96.    {
97.      String beanName = (String) e.getKey();
98.      String eventCode = (String) e.getValue();
99.      source.append(beanName
100.       + ".addActionListener(new java.awt.event.ActionListener() {");
101.     source.append("public void actionPerformed(java.awt.event.ActionEvent event) {");
102.     source.append(eventCode);
103.     source.append("} } );");
104.   }
105.   source.append("} }");
106.   return source;
107.  }
108. }
```

Листинг 11.7. Содержимое файла `MapClassLoader.java`

```
1.  import java.util.*;
2.  /**
3.   * Загрузчик классов, который загружает классы из таблицы, где в роли
4.   * ключей выступают имена классов, а в роли значений - массивы байт-кодов.
5.   * @version 1.00 2007-11-02
6.   * @author Cay Horstmann
7.   */
8.  public class MapClassLoader extends ClassLoader
9.  {
10.   public MapClassLoader(Map<String, byte[]> classes)
11.   {
12.     this.classes = classes;
13.   }
```

```
14.    protected Class<?> findClass(String name) throws ClassNotFoundException
15.    {
16.      byte[] classBytes = classes.get(name);
17.      if (classBytes == null) throw new ClassNotFoundException(name);
18.      Class<?> cl = defineClass(name, classBytes, 0, classBytes.length);
19.      if (cl == null) throw new ClassNotFoundException(name);
20.      return cl;
21.    }
22.    private Map<String, byte[]> classes;
23.  }
```

Использование аннотаций

Аннотациями (annotations) называются дескрипторы, которые разработчики вставляют в свой исходный код для того, чтобы та или иная утилита могла обрабатывать их. Эти утилиты могут работать как на уровне исходного кода, так и на уровне файлов классов, в которые компилятор помещает аннотации.

Аннотации не влияют на способ компиляции программ. Компилятор Java генерирует одинаковые инструкции виртуальной машины как с, так и без аннотаций.

Для извлечения пользы из аннотаций необходимо выбирать *средство обработки* (processing tool). В код вставляются аннотации, понятные определенному средству обработки, которое потом всегда и применяется для их расшифровки.

У аннотаций существует масса сфер применения, и такая их универсальность сначала может вызывать путаницу. Ниже перечислены некоторые из этих сфер:

- автоматическая генерация вспомогательных файлов, вроде файлов дескрипторов развертывания или классов информации о bean-компонентах;
- автоматическая генерации кода для тестирования, регистрации, семантической обработки транзакций и т.д.

Мы начнем обсуждение аннотаций с базовых концепций и продемонстрируем их применение на практике с помощью конкретного примера: в частности, мы пометим методы как слушатели событий для компонентов AWT и покажем обработчик аннотаций, анализирующий аннотации и подключающий слушателей. Далее мы подробно рассмотрим синтаксические правила. И, наконец, напоследок мы предложим два более сложных примера обработки аннотаций: первый с обработкой аннотаций на уровне исходного кода, а второй с обработкой аннотаций на уровне файлов классов за счет использования технологии Apache Bytecode Engineering Library и вставки в аннотированные методы дополнительных байт-кодов.

Ниже приведен пример простой аннотации:

```
public class MyClass
{
  . . .
  @Test public void checkRandomInsertions()
}
```

Аннотация `@Test` аннотирует метод `checkRandomInsertions`.

В Java аннотация применяется подобно *модификатору* (modifier) и помещается перед аннотируемым элементом *без точки с запятой*. (Модификатор — это ключевое слово вроде `public` или `static`.) Перед именем каждой аннотации ставится символ `@`, подобно тому, как это делается в случае комментариев Javadoc. Однако комментарии Javadoc размещаются внутри разделителей `/**...*/`, тогда как аннотации являются частью кода.

Сама по себе аннотация `@Test` ничего не делает. Для того чтобы приносить пользу, ей необходим инструмент. Например, инструмент тестирования JUnit 4 (доступный на сайте `http://junit.org`) умеет вызвать все помеченные аннотацией `@Test` методы при тестировании класса. Другой инструмент может удалять все тестовые методы из файла класса, так чтобы они не поставлялись с программой уже после прохождения ею тестирования.

Аннотации могут определяться с *элементами*, например:

```
@Test(timeout="10000")
```

Эти элементы могут обрабатываться инструментами, читающими аннотации. Элементы могут выглядеть и по-другому; более подробно о них будет рассказываться позже в этой главе.

Помимо методов снабжаться аннотациями также могут классы, поля и локальные переменные — аннотация может размещаться во всех тех же местах, что модификатор вроде `public` или `static`.

Каждая аннотация должна определяться с помощью *аннотационного интерфейса* (annotation interface). Методы этого интерфейса должны соответствовать элементам определяемой им аннотации. Например, JUnit-аннотация `Test` определяется с помощью такого интерфейса:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test
{
  long timeout() default 0L;
  . . .
}
```

Объявление `@interface` создает фактический интерфейс Java, а обрабатывающие аннотации инструменты получают объекты, которые реализуют уже аннотационный интерфейс. Для извлечения элемента `timeout` конкретно аннотации `Test`, например, инструмент будет вызывать метод `timeout`.

Аннотации `Target` и `Retention` являются *мета-аннотациями* (meta-annotations). Они аннотируют аннотацию `Target`, помечая ее как аннотацию, которая может применяться только к методам и которая должна сохраняться при загрузке файла класса в виртуальную машину. Более подробно о них будет рассказываться чуть позже в этой главе, в разделе “Мета-аннотации”.

В этом разделе вы ознакомились с базовыми концепциями программных метаданных и аннотаций. В следующем разделе будет рассмотрен конкретный пример обработки аннотаций.

Пример: аннотирование обработчиков событий

Одной из наиболее утомительных задач в программировании пользовательских интерфейсов является присоединение слушателей к источникам событий. Многие слушатели имеют такой вид:

```
myButton.addActionListener(new
  ActionListener()
  {
    public void actionPerformed(ActionEvent event)
    {
      doSomething();
    }
});
```

В этом разделе во избежание подобной нудной работы создается аннотация. Выглядит эта аннотация следующим образом:

```
@ActionListenerFor(source="myButton") void doSomething() { . . . }
```

Программисту больше не нужно делать вызовы к `addActionListener`. Вместо этого каждый метод просто помечается аннотацией. В листинге 11.8 показан реализованный заново вместе с такими аннотациями класс `ButtonFrame`, который приводился в главе 8 первого тома этой книги.

Конечно, еще также потребуется реализовать и аннотационный интерфейс. Весь необходимый для этого код показан в листинге 11.9.

Листинг 11.8. Содержимое файла `ButtonFrame.java`

```
1.  import java.awt.*;
2.  import javax.swing.*;
3.  /**
4.   * Фрейм с кнопочной панелью
5.   * @version 1.00 2004-08-17
6.   * @author Cay Horstmann
7.   */
8.  public class ButtonFrame extends JFrame
9.  {
10.   public ButtonFrame()
11.   {
12.     setTitle("ButtonTest");
13.     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
14.     panel = new JPanel();
15.     add(panel);
16.
17.     yellowButton = new JButton("Yellow");
18.     blueButton = new JButton("Blue");
19.     redButton = new JButton("Red");
20.
21.     panel.add(yellowButton);
22.     panel.add(blueButton);
23.     panel.add(redButton);
24.
25.     ActionListenerInstaller.processAnnotations(this);
26.   }
27.
28.   @ActionListenerFor(source = "yellowButton")
29.   public void yellowBackground()
30.   {
31.     panel.setBackground(Color.YELLOW);
32.   }
33.
34.   @ActionListenerFor(source = "blueButton")
35.   public void blueBackground()
36.   {
37.     panel.setBackground(Color.BLUE);
38.   }
39.
40.   @ActionListenerFor(source = "redButton")
41.   public void redBackground()
42.   {
43.     panel.setBackground(Color.RED);
44.   }
```

```
45.    public static final int DEFAULT_WIDTH = 300;
46.    public static final int DEFAULT_HEIGHT = 200;
47.
48.    private JPanel panel;
49.    private JButton yellowButton;
50.    private JButton blueButton;
51.    private JButton redButton;
52.  }
```

Листинг 11.9. Содержимое файла `ActionListenerFor.java`

```
1.   import java.lang.annotation.*;
2.
3.   /**
4.    * @version 1.00 2004-08-17
5.    * @author Cay Horstmann
6.    */
7.
8.   @Target(ElementType.METHOD)
9.   @Retention(RetentionPolicy.RUNTIME)
10.  public @interface ActionListenerFor
11.  {
12.    String source();
13.  }
```

Разумеется, сами по себе аннотации ничего не делают. Они просто хранятся в исходном файле. Компилятор помещает их в файл классов, а виртуальная машина загружает их. Следовательно, далее требуется какой-то механизм для выполнения их анализа и установки слушателей действий. Эта обязанность возлагается на класс `ActionListenerInstaller`. Конструктор `ButtonFrame` вызывает

```
ActionListenerInstaller.processAnnotations(this);
```

Статический метод `processAnnotations` перечисляет все методы полученного им объекта. Далее для каждого метода он извлекает объект аннотации `ActionListenerFor` и обрабатывает его:

```
Class<?> cl = obj.getClass();
for (Method m : cl.getDeclaredMethods())
{
  ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
  if (a != null) . . .
}
```

Здесь 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. Мы извлекаем его путем вызова метода `source`, а затем выполняем поиск соответствующего ему поля:

```
String fieldName = a.source();
Field f = cl.getDeclaredField(fieldName);
```

Это демонстрирует ограниченность нашей аннотации. Исходный элемент должен обязательно представлять собой имя поля. Представлять локальную переменную он не может.

Остальная часть кода носит довольно технический характер. Для каждого аннотированного метода создается прокси-объект, реализующий интерфейс `ActionListener`

с методом `actionPerformed`, который вызывает данный аннотированный метод. (Более подробную информацию о прокси-объектах можно найти в главе 6 первого тома этой книги.) Детали роли не играют. Главным здесь является то, что функциональность аннотаций была установлена с помощью метода `processAnnotations`.

Весь процесс обработки аннотаций схематично показан на рис. 11.1.



Рис. 11.1. Обработка аннотаций во время выполнения

В данном примере аннотации обрабатывались во время выполнения. Однако они также могли обрабатываться и на уровне исходного кода: генератор исходного кода мог бы генерировать код для добавления слушателей. В качестве альтернативного варианта аннотации еще также могли бы обрабатываться и на уровне байт-кода: редактор байт-кода мог бы вставлять вызовы `addActionListener` в конструктор фрейма. Звучит довольно сложно, но в настоящее время доступны библиотеки, которые делают выполнение этой задачи относительно простым процессом. Удостовериться в этом можно будет на конкретном примере чуть позже в этой главе, в разделе “Разработка байт-кода”.

Пример, приведенный в этом разделе, ни в коей мере не является рекомендуемым серьезным средством для разработчиков пользовательских интерфейсов. Подход с применением утилиты для добавления слушателей может быть для программистов таким же удобным, как и подход с добавлением аннотаций. (На самом деле, класс `java.beans.EventHandler` как раз такой подход и пытается реализовать. Его легко сделать по-настоящему полезным путем предоставления метода, добавляющего обработчик событий, вместо того, чтобы просто создавать его.)

Однако этот пример иллюстрирует механику процесса аннотирования программы и анализа аннотаций. Его рассмотрение должно сделать читателя более подготовленным к изучению материала следующих разделов, в которых приводится подробнейшее описание синтаксиса аннотаций.

Листинг 11.10. Содержимое файла `ActionListenerInstaller.java`

```
1.  import java.awt.event.*;
2.  import java.lang.reflect.*;
3.
4.  /**
5.   * @version 1.00 2004-08-17
6.   * @author Cay Horstmann
7.   */
8.  public class ActionListenerInstaller
9.  {
```

```
10.   /**
11.    * Обрабатывает все аннотации ActionListenerFor в заданном объекте.
12.    * @param obj Объект, методы которого
13.    * могут содержать аннотации ActionListenerFor.
14.    */
15.   public static void processAnnotations(Object obj)
16.   {
17.     try
18.     {
19.       Class<?> cl = obj.getClass();
20.       for (Method m : cl.getDeclaredMethods())
21.       {
22.         ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
23.         if (a != null)
24.         {
25.           Field f = cl.getDeclaredField(a.source());
26.           f.setAccessible(true);
27.           addListener(f.get(obj), obj, m);
28.         }
29.       }
30.     }
31.     catch (Exception e)
32.     {
33.       e.printStackTrace();
34.     }
35.   }
36.
37.   /**
38.    * Добавляет слушателя действий, который вызывает заданный метод.
39.    * @param source Источник событий, к которому добавляется
40.    * слушатель действий.
41.    * @param param Неявный параметр метода, который вызывает слушатель.
42.    * @param m Метод, который вызывает слушатель.
43.    */
44.   public static void addListener(Object source, final Object param, final Method m)
45.     throws NoSuchMethodException, IllegalAccessException, InvocationTargetException
46.   {
47.     InvocationHandler handler = new InvocationHandler()
48.     {
49.       public Object invoke(Object proxy, Method mm, Object[] args) throws Throwable
50.       {
51.          return m.invoke(param);
52.       }
53.     };
54.
55.     Object listener = Proxy.newProxyInstance(null,
56.         new Class[] { java.awt.event.ActionListener.class }, handler);
57.
58.     Method adder =
59.        source.getClass().getMethod("addActionListener", ActionListener.class);
60.
61.     adder.invoke(source, listener);
62.   }
63. }
```

API java.lang.AnnotatedElement 5.0

- `boolean isAnnotationPresent(Class<? extends Annotation> annotationType)`
  Возвращает `true`, если данный элемент имеет аннотацию заданного типа.
- `<T extends Annotation> T getAnnotation(Class<T> annotationType)`
  Получает аннотацию заданного типа или `null`, если у данного элемента нет такой аннотации.
- `Annotation[] getAnnotations()`
  Получает все аннотации, которые доступны для данного элемента, включая унаследованные. Если доступных аннотаций нет, возвращает массив нулевой длины.
- `Annotation[] getDeclaredAnnotations()`
  Получает все аннотации, которые были объявлены для данного элемента, исключая унаследованные. Если доступных аннотаций нет, возвращает массив нулевой длины.

Синтаксис аннотаций

В этом разделе рассказывается обо всем, что необходимо знать о синтаксисе аннотаций. Аннотация определятся с помощью аннотационного интерфейса:

```
модификаторы @interface ИмяАннотации
{
  объявление элемента 1
  объявление элемента 2
  . . .
}
```

Объявление каждого элемента выглядит либо так:

```
тип имяЭлемента();
```

либо так:

```
тип имяЭлемента() default значение_по_умолчанию;
```

Например, следующая аннотация имеет два элемента — `assignedTo` и `severity`:

```
public @interface BugReport
{
  String assignedTo() default "[none]";
  int severity() = 0;
}
```

Каждая аннотация имеет такой формат:

```
@ ИмяАннотации(имяЭлемента1=значение1, имяЭлемента2=значение2, . . .)
```

Например:

```
@BugReport(assignedTo="Harry", severity=10)
```

Порядок следования элементов роли не играет. Аннотация

```
@BugReport(severity=10, assignedTo="Harry")
```

абсолютно идентична приведенной выше.

Стандартное значение (`default`) объявления используется в случае не указания значения для элемента. Например, рассмотрим следующую аннотацию:

```
@BugReport(severity=10)
```

Здесь значением элемента `assignedTo` является строка `"[none]"`.



Значения по умолчанию не хранятся вместе с аннотацией; вместо этого они вычисляются динамическим образом. Например, если заменить значение по умолчанию для элемента `assignedTo` на значение `"[]"` и скомпилировать интерфейс `BugReport` заново, аннотация `@BugReport(severity=10)` будет иметь новое значение по умолчанию, даже в тех файлах классов, которые были скомпилированы до изменения значения по умолчанию.

Два специальных сокращения позволяют упрощать аннотации.

Если никаких элементов не указано, либо из-за того, что у аннотации нет таковых, либо из-за того, что для всех из них используются значения по умолчанию, круглые скобки не потребуются. Например, аннотация

```
@BugReport
```

будет иметь абсолютно точно такой же эффект, как и аннотация

```
@BugReport(assignedTo="[none]", severity=0)
```

Аннотации такого типа называются *маркерными аннотациями* (marker annotation).

Другим видом сокращения являются *однозначные аннотации* (single value annotation). Если элемент имеет специальное имя `value` и больше никаких элементов не указано, тогда имя элемента и символ = могут опускаться. Например, если бы мы в предыдущем разделе определили интерфейс аннотаций `ActionListenerFor` так:

```
public @interface ActionListenerFor
{
  String value();
}
```

тогда мы могли бы представлять аннотации не в виде `@ActionListenerFor(value="yellowButton")`, а в следующим образом:

```
@ActionListenerFor("yellowButton")
```

Все интерфейсы аннотаций неявно расширяют интерфейс `java.lang.annotation.Annotation`. Этот интерфейс является обычным интерфейсом, а *не* интерфейсом аннотаций. Предоставляемые им методы можно найти в конце этого раздела, во врезках API.

Самостоятельно расширять интерфейсы аннотаций нельзя. Другими словами, все интерфейсы аннотаций могут только непосредственно расширять интерфейс `java.lang.annotation.Annotation`.

Классы, реализующие интерфейсы аннотаций, никогда не предоставляются. Вместо этого виртуальная машина сама генерирует подходящие прокси-классы и объекты по мере необходимости. Например, при запрашивании аннотации `ActionListenerFor` виртуальная машина будет выполнять приблизительно такую операцию:

```
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 конструкций `throws`, но не могут быть обобщенными (generic).

Типом элемента аннотации может быть:

- примитивный тип (int, short, long, byte, char, double, float или boolean);
- String;
- Class (с каким-то необязательным параметром вроде Class<? extends MyClass>);
- тип enum;
- тип аннотации;
- массив перечисленных выше типов (массив массивов не является допустимым типом элемента).

Ниже приведены примеры допустимых объявлений типов:

```
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();
}
```

Поскольку аннотации оцениваются компилятором, значения всех элементов должны обязательно представлять собой статические константы. Например:

```
@BugReport(showStopper=true, assignedTo="Harry", testCase=MyTestCase.class,
  status=BugReport.Status.CONFIRMED, . . .)
```



Элемент аннотации никогда не может устанавливаться в null. Даже значение по умолчанию null не допускается. Это может вызвать некоторые трудности на практике. Придется выбирать других кандидатов на значение по умолчанию, вроде "" или Void.class.

Если значением элемента является массив, его значения заключаются в круглые скобки, подобно тому, как показано ниже:

```
@BugReport(. . ., reportedBy={"Harry", "Carl"})
```

Если элемент имеет одно единственное значение, фигурные скобки могут опускаться:

```
@BugReport(. . ., reportedBy="Joe") // допускается, идентично {"Joe"}
```

Поскольку элементом аннотации может быть еще какая-нибудь аннотация, это дает возможность создавать довольно сложные аннотации. Например:

```
@BugReport(ref=@Reference(id="3352627"), . . .)
```



Ввод циклических зависимостей в аннотациях является ошибкой. Например, из-за того, что BugReport имеет элемент типа аннотации Reference, аннотация Reference не может иметь элемент типа BugReport.

Добавлять аннотации можно к следующим элементам:

- пакеты;
- классы (enum включительно);
- интерфейсы (включая интерфейсы аннотаций);

- методы;
- конструкторы;
- поля экземпляров (константы `enum` включительно);
- локальные переменные;
- переменные параметров.

Аннотации, предназначенные для локальных переменных, однако, могут обрабатываться только на уровне исходного кода. Файлы классов не описывают локальных переменных. Из-за этого при компиляции класса аннотации всех локальных переменных отбрасываются. Подобным образом обстоит и дело с аннотациями для пакетов: они тоже не сохраняются за пределами уровня исходного кода.



Аннотирование пакетов осуществляется в файле `package-info.java`, в котором может содержаться только оператор пакета с предшествующими ему аннотациями.

У каждого элемента может быть множество аннотаций, главное, чтобы они относились к разным типам. Использовать один и тот же тип аннотации более одного раза при аннотировании определенного элемента нельзя. Например, следующий код

```
@BugReport(showStopper=true, reportedBy="Joe")
@BugReport(reportedBy={"Harry", "Carl"})
void myMethod()
```

во время компиляции будет приводить к появлению ошибки. Если это является проблемой, можно создать аннотацию со значением, представляющим собой массив более простых аннотаций:

```
@BugReports({
  @BugReport(showStopper=true, reportedBy="Joe"),
  @BugReport(reportedBy={"Harry", "Carl"}))
void myMethod()
```

API java.lang.annotation.Annotation 5.0

- `Class<? extends Annotation> annotationType()`

  Возвращает объект `Class`, который представляет аннотационный интерфейс данного объекта аннотации. Обратите внимание на то, что вызов метода `getClass` на объекте аннотации приводил бы к возвращению фактического класса, а не интерфейса.

- `boolean equals(Object other)`

  Возвращает `true`, если `other` представляет объект, реализующий тот же аннотационный интерфейс, что и данный объект аннотации, и если все элементы этого объекта и `other` равны друг другу.

- `int hashCode()`

  Возвращает хеш-код, который является совместимым с методом `equals` и получается из имени аннотационного интерфейса и значений его элементов.

- `String toString()`

  Возвращает строковое представление, содержащее название аннотационного интерфейса и значения элементов, например, `@BugReport(assignedTo=[none], severity=0)`.

Стандартные аннотации

В Java SE имеется ряд уже определенных интерфейсов аннотаций внутри таких пакетов, как `java.lang`, `java.lang.annotation` и `javax.annotation`. Четыре из них представляют собой мета-аннотации, которые лишь описывают поведение аннотационных интерфейсов. Остальные являются обычными аннотациями, которые разработчики могут применять для аннотирования элементов в своем исходном коде. Все эти аннотации вкратце перечислены в табл. 11.2 и более подробно будут описаны в двух следующих разделах.

Таблица 11.2. Стандартные аннотации

Аннотационный интерфейс	Подходит для	Предназначение
`Deprecated`	всего	Помечает элемент как не рекомендуемый.
`SuppressWarnings`	всего, кроме пакетов и аннотаций	Подавляет предупреждения указанного типа.
`Override`	методов	Проверяет, чтобы данный метод переопределял метод суперкласса.
`PostConstruct` `PreDestroy`	методов	Помеченный метод должен вызываться сразу же после создания или непосредственно перед удалением.
`Resource`	классов, интерфейсов, методов, полей	В случае класса или интерфейса помечает его как ресурс, подлежащий использованию в каком-то другом месте. В случае метода или поля помечает его как подлежащий “внесению”.
`Resources`	классов и интерфейсов	Массив ресурсов.
`Generated`	всего	Помечает элемент как исходный код, который был сгенерирован какой-то утилитой.
`Target`	аннотаций	Указывает элементы, к которым может применяться данная аннотация.
`Retention`	аннотаций	Указывает, как долго должна сохраняться данная аннотация.
`Documented`	аннотаций	Указывает, что данная аннотация должна включаться в документацию аннотируемых элементов.
`Inherited`	аннотаций	Указывает, что данная аннотация, в случае применения к классу, будет автоматически наследоваться всеми его подклассами.

Аннотации для компиляции

Аннотация `@Deprecated` может присоединяться к любым элементам, использование которых более не поощряется. Тогда в случае применения не рекомендуемого элемента компилятор будет выдавать соответствующее предупреждение. Эта аннотация имеет то же предназначение, что и Javadoc-дескриптор `@deprecated`.

Аннотация `@SuppressWarnings` указывает компилятору подавлять предупреждения определенного типа, например:

```
@SuppressWarnings("unchecked")
```

Аннотация `@Override` применяется только к методам. Компилятор проверяет, чтобы метод с такой аннотацией действительно переопределял метод из суперкласса. Например, в случае использования следующего объявления:

```
public MyClass
{
  @Override public boolean equals(MyClass other);
  . . .
}
```

компилятор выдаст ошибку, поскольку метод `equals` *не* переопределяет метод `equals` класса `Object`. Этот метод имеет параметр типа `Object`, а не `MyClass`.

Аннотация `@Generated` предназначена для использования инструментами генерации кода. Любой генерируемый исходный код может снабжаться аннотациями для того, чтобы он отличался от кода, предоставляемого программистом. Например, редактор кода может скрывать сгенерированный код, а генератор кода — удалять более старые версии сгенерированного кода. Каждая аннотация должна обязательно содержать уникальный идентификатор для генератора кода. Строка даты (в формате ISO8601) и строка комментария являются необязательными. Например:

```
@Generated("com.horstmann.beanproperty", "2008-01-04T12:08:56.235-0700");
```

Аннотации для управления ресурсами

Аннотации `@PostConstruct` и `@PreDestroy` применяются в таких средах, которые умеют управлять жизненным циклом объектов, например, в Web-контейнерах и серверах приложений. Помечаемые с помощью таких аннотаций методы должны вызываться либо сразу же после создания объекта, либо непосредственно перед его удалением.

Аннотация `@Resource` предназначена для внедрения ресурсов. Например, рассмотрим Web-приложение, получающее доступ к базе данных. Разумеется, информация о получении доступа к базе данных не должна жестко кодироваться в приложении. Вместо этого у Web-контейнера должен быть какой-то пользовательский интерфейс для установки параметров подключения и JNDI-имя для источника данных. Сослаться на источник данных в Web-приложении можно следующим образом:

```
@Resource(name="jdbc/mydb")
private DataSource source;
```

В результате при создании объекта, содержащего это поле, контейнер будет автоматически "внедрять" ссылку на источник данных

Мета-аннотации

Мета-аннотация `@Target` применяется к аннотации, ограничивая те элементы, к которым она должна применяться. Например:

```
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface BugReport
```

В табл. 11.3 перечислены все возможные значения. Они относятся к перечислимому типу `ElementType`. Указывать можно любое количество типов элементов, заключая их в фигурные скобки.

Аннотация без ограничения `@Target` может применяться к любому элементу. Компилятор проверяет, чтобы аннотация применялась только к разрешенному элементу.

Таблица 11.3. Типы элементов для аннотации `@Target`

Тип элемента	Аннотация применяется к:
`ANNOTATION_TYPE`	объявлениям типов аннотаций
`PACKAGE`	ракетам
`TYPE`	классам (включая `enum`) и интерфейсам (включая типы аннотаций)
`METHOD`	методам
`CONSTRUCTOR`	конструкторам
`FIELD`	полям (константам `enum` включительно)
`PARAMETER`	параметрам методов или конструкторов
`LOCAL_VARIABLE`	локальным переменным

Например, в случае применения `@BugReport` к полю, компилятор выдаст во время компиляции соответствующую ошибку.

Мета-аннотация `@Retention` указывает, насколько долго должна сохраняться аннотация. Задается, максимум, лишь одно из приведенных в табл. 11.4 значений. Значением по умолчанию является `RetentionPolicy.CLASS`.

Таблица 11.4. Политики сохранности для аннотации `@Retention`

Политика сохранности	Описание
`SOURCE`	Аннотации не включаются в файлы классов.
`CLASS`	Аннотации включаются в файлы классов, но виртуальной машине не нужно загружать их.
`RUNTIME`	Аннотации включаются в файлы классов и загружаются виртуальной машиной. Они доступны через API-интерфейс рефлексии.

В листинге 11.9 аннотация `@ActionListenerFor` была объявлена со значением `RetentionPolicy.RUNTIME`, поскольку там для обработки аннотаций применялась рефлексия. В следующих двух разделах будут приведены примеры обработки аннотаций на уровне исходного кода и на уровне файлов классов.

Мета-аннотация `@Documented` дает подсказку о средствах документирования вроде `Javadoc`. Документируемые аннотации должны применяться точно так же, как и другие модификаторы, вроде `protected` или `static`, в целях документирования. Использование остальных аннотаций в документации не фиксируется. Например, предположим, что `@ActionListenerFor` объявляется как документируемая аннотация:

```
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ActionListenerFor
```

Тогда документация каждого аннотированного метода будет содержать такую аннотацию, как показано на рис. 11.2.

Если аннотация является временной (как, например, `@BugReport`), тогда документировать ее использование, пожалуй, не стоит.



Применение аннотации к самой себе является вполне допустимым. Например, аннотация `@Documented` сама аннотируется как `@Documented`. Поэтому документация Javadoc по аннотациям и показывает, являются они документируемыми или нет.

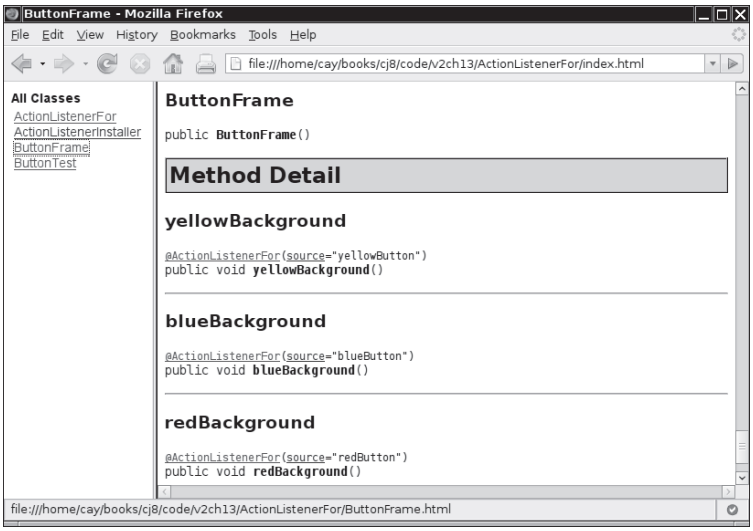


Рис. 11.2. Документируемые аннотации

Мета-аннотация `@Inherited` применяется только к аннотациям классов. Когда в класс добавляется наследуемая аннотация, все его подклассы автоматически получают точно такую же аннотацию. Это позволяет легко создавать аннотации, работающие точно так же, как и маркерные интерфейсы наподобие `Serializable`.

На самом деле аннотация `@Serializable` будет более уместной, чем маркерные интерфейсы `Serializable` без единого метода. Класс является сериализируемым из-за наличия на этапе выполнения поддержки для считывания и записи его полей, а не из-за каких-то там принципов объектно-ориентированной разработки. Аннотация описывает этот факт гораздо лучше, чем это делает наследование интерфейсов. Конечно, интерфейс `Serializable` появился еще в JDK 1.1, то есть гораздо раньше, чем аннотации.

Предположим, что для указания того, что объекты класса могут сохраняться в базе данных, определяется наследуемая аннотация `@Persistent`. Тогда все подклассы этого постоянного класса будут автоматически аннотироваться как постоянные:

```
@Inherited @Persistent { }
@Persistent class Employee { . . . }
class Manager extends Employee { . . . } // тоже @Persistent
```

При выполнении поиска объектов, подлежащих хранению в базе данных, механизм сохранения будет обнаруживать как объекты `Employee`, так и объекты `Manager`.

Обработка аннотаций на уровне исходного кода

Одной из сфер применения аннотаций является автоматическая генерация “побочных файлов” (side files), содержащих дополнительную информацию о программах. В прошлом, когда в ходу была печально известная версия Java Enterprise Edition, программистам доводилось возиться с огромным количеством шаблонного кода. Сегодня, в версии Java EE 5 доступны аннотации, которые позволяют значительно упрощать модель программирования.

В этом разделе демонстрируется упомянутый подход на примере простой программы, которая автоматически генерирует классы информации о bean-компонентах. Свойства bean-компонентов снабжаются аннотациями, после чего запускается утилита, которая выполняет синтаксический разбор исходного файла, анализирует все аннотации и выдает исходный файл класса информации о bean-компоненте.

Вспомним из главы 8, что класс информации о bean-компоненте описывает bean-компонент более точно, чем это способен делать автоматический процесс самоанализа. Он дает перечень всех свойств bean-компонента. Эти свойства могут обладать необязательными редакторами свойств. Типичным тому примером является приводившийся в главе 8 класс `ChartBeanBeanInfo`.

Во избежание утомительной работы по написанию классов информации о bean-компонентах в демонстрируемом здесь примере добавляется аннотация `@Property`. Эта аннотация может добавляться как к методу извлечения, так и к методу установки свойства, подобно тому, как показано ниже:

```
@Property String getTitle() { return title; }
```

или

```
@Property(editor="TitlePositionEditor")
public void setTitlePosition(int p) { titlePosition = p; }
```

В листинге 11.11 содержится определение аннотации `@Property`. Обратите внимание на то, что эта аннотация имеет политику сохранности со значением `SOURCE`. Это означает, что она может анализироваться только на уровне исходного кода. Она не включается в файлы классов и не доступна во время рефлексии.

Листинг 11.11. Содержимое файла `Property.java`

```
1.  package com.horstmann.annotations;
2.  import java.lang.annotation.*;
3.  @Documented
4.  @Target(ElementType.METHOD)
5.  @Retention(RetentionPolicy.SOURCE)
6.  public @interface Property
7.  {
8.    String editor() default "";
9.  }
```



Имело бы смысл объявить элемент `editor` с типом `Class`. Однако 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 используется строка.

Далее для автоматической генерации класса информации о bean-компонентах для класса с именем *`BeanClass`* выполняются следующие операции.

1. Записывается исходный файл *`BeanClass`*`BeanInfo.java`. Объявляется класс *`BeanClass`*`BeanInfo` для расширения `SimpleBeanInfo`, и переопределяется метод `getPropertyDescriptors`.
2. Для каждого аннотированного метода восстанавливается имя свойства за счет отсечения префикса `get` или `set` и преобразования в оставшейся части заглавных букв в прописные.
3. Для каждого свойства пишется оператор по созданию объекта `PropertyDescriptor`.

4. Если у свойства имеется редактор, пишется вызов метода `setPropertyEditorClass`.
5. Записывается код для возвращения массива дескрипторов всех свойств.

Например, аннотация

```
@Property(editor="TitlePositionEditor")
public void setTitlePosition(int p) { titlePosition = p; }
```

в классе `ChartBean` преобразуется следующим образом:

```
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,
   . . .
  }
 }
}
```

(Шаблонный код представлен более светлым цветом.)

Все это делается довольно легко, при условии, что удается установить местонахождение всех методов, которые были помечены аннотацией `@Property`.

В версии Java SE 6 *обработчики аннотаций* (annotation processors) могут добавляться к компилятору Java. (В Java SE 5 для этой цели применялась отдельная утилита под названием `apt`). Инициируется обработка аннотаций с помощью следующей команды:

```
javac -processor Имя_класса_обработчика1, Имя_класса_обработчика2,... исходные_файлы
```

Компилятор обнаруживает аннотации исходных файлов. Далее он выбирает обработчиков аннотаций, которые следует применить. Каждый обработчик аннотаций выполняется в свою очередь. Если обработчик аннотаций создает новый исходный файл, тогда процесс повторяется. Если цикл обработки не приводит к созданию больше никаких исходных файлов, тогда выполняется компиляция всех исходных файлов. На рис. 11.3 показана схема обработки аннотаций `@Property`.

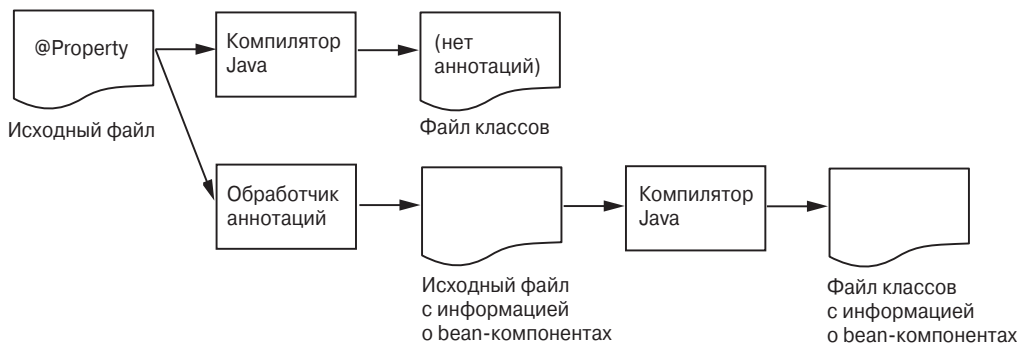


Рис. 11.3. Обработка аннотаций на уровне исходных файлов

Рассматривать более подробно API-интерфейс обработки аннотаций мы не будем, но программа в листинге 11.12 позволит получить общее представление о его возможностях.

Обработчик аннотаций реализует интерфейс `Processor`, обычно за счет расширения класса `AbstractProcessor`. Разработчику необходимо указывать, какие аннотации поддерживает его обработчик. Из-за того, что разработчики API просто обожают аннотации, они часто используют их и для такой цели:

```
@SupportedAnnotationTypes("com.horstmann.annotations.Property")
public class BeanInfoAnnotationProcessor extends AbstractProcessor
```

Обработчик может запрашивать аннотации конкретных типов, групповые символы вроде `com.horstmann.*` (все аннотации из пакета `com.horstmann` или любого из его подпакетов) и даже `"*"` (все аннотации).

У `BeanInfoAnnotationProcessor` имеется один единственный общедоступный метод — `process`, который вызывается для каждого файла. Этот метод принимает два параметра — набор аннотаций, которые обрабатываются в данном цикле, и ссылку `RoundEnv`, в которой содержится информация о текущем цикле обработки.

В рамках метода `process` осуществляется проход по всем аннотированным методам. Для каждого из этих методов получается имя свойства путем отбрасывания префикса `get`, `set` или `is` и замены следующей за ним буквы прописной. В общем, структура этого кода выглядит так:

```
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(имя_свойства, e.getAnnotation(Property.class));
      }
      запись исходного файла информации о bean-компоненте
   }
   return true;
}
```

Метод `process` должен возвращать `true`, если он *предъявляет требования на* все представленные ему аннотации, то есть если эти аннотации больше не должны передаваться дальше никаким 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 выходных 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 для того, чтобы они могли обрабатываться в последующих циклах обработки.

Когда обработчик аннотаций обнаруживает ошибку, он использует `Messager` для взаимодействия с пользователем. Например, в случае, если метод был снабжен аннотацией `@Property`, но его имя не начинается с `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 для этой книги предлагается аннотированный файл `ChartBean.java`:

Скомпилируйте обработчик аннотаций:

```
javac BeanInfoAnnotationProcessor.java
```

Затем выполните следующую команду:

```
javac -processor BeanInfoAnnotationProcessor com/horstmann/corejava/ChartBean.java
```

и посмотрите на автоматически сгенерированный файл `ChartBeanBeanInfo.java`.

Чтобы увидеть, как процесс обработки аннотаций выглядит в действии, добавьте к команде `javac` командную опцию `XprintRounds` и тогда вы получите следующий вывод:

```
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, как инструменты могут собирать аннотации исходных файлов для генерации других файлов. Генерируемые таким образом файлы необязательно должны представлять собой исходные файлы. Обработчики аннотаций могут отдавать предпочтение генерации дескрипторов XML, файлов свойств, сценариев оболочки, документации HTML и т.д.



Поступали предложения использовать аннотации для устранения необходимости в выполнении даже еще большего количества нудной работы. Ну, разве было бы не замечательно, если бы тривиальные методы извлечения (getter) и установки (setter) генерировались автоматически? Например, аннотация

```
@Property private String title;
```

могла бы автоматические генерировать такие методы, как:

```
public String getTitle() { return title; }
public void setTitle(String title) { this.title = title; }
```

Такие методы, однако, необходимо добавлять в *один и тот же* класс. А это требует выполнения редактирования исходного файла, а не только генерации еще одного файла, что выходит за рамки возможностей обработчиков аннотаций. Можно было бы создать другой инструмент для данной цели, но этот инструмент тогда бы уже выходил за пределы предназначения аннотаций. Предназначением аннотации является представление описания *об* элементе кода, а не директивы для добавления или изменения кода.

Листинг 11.12. Содержимое файла `BeanInfoAnnotationFactory.java`

```
1.  import java.beans.*;
2.  import java.io.*;
3.  import java.util.*;
4.  import javax.annotation.processing.*;
5.  import javax.lang.model.*;
```

```
6.  import javax.lang.model.element.*;
7.  import javax.tools.*;
8.  import javax.tools.Diagnostic.*;
9.  import com.horstmann.annotations.*;
10.
11. /**
12.  * Этот класс является обработчиком, который
13.  * анализирует аннотации Property.
14.  * @version 1.10 2007-10-27
15.  * @author Cay Horstmann
16.  */
17. @SupportedAnnotationTypes("com.horstmann.annotations.Property")
18. @SupportedSourceVersion(SourceVersion.RELEASE_6)
19. public class BeanInfoAnnotationProcessor extends AbstractProcessor
20. {
21.   @Override
22.   public boolean process(Set<? extends TypeElement> annotations,
23.                          RoundEnvironment roundEnv)
24.   {
25.     for (TypeElement t : annotations)
26.     {
27.       Map<String, Property> props = new LinkedHashMap<String, Property>();
28.       String beanClassName = null;
29.       for (Element e : roundEnv.getElementsAnnotatedWith(t))
30.       {
31.         String mname = e.getSimpleName().toString();
32.         String[] prefixes = { "get", "set", "is" };
33.         boolean found = false;
34.         for (int i = 0; !found && i < prefixes.length; i++)
35.           if (mname.startsWith(prefixes[i]))
36.           {
37.             found = true;
38.             int start = prefixes[i].length();
39.             String name = Introspector.decapitalize(mname.substring(start));
40.             props.put(name, e.getAnnotation(Property.class));
41.           }
42.
43.         if (!found) processingEnv.getMessager().printMessage(Kind.ERROR,
44.           "@Property must be applied to getXxx, setXxx, or isXxx method", e);
45.         else if (beanClassName == null)
46.           beanClassName = ((TypeElement) e.getEnclosingElement()).getQualifiedName()
47.             .toString();
48.       }
49.       try
50.       {
51.         if (beanClassName != null) writeBeanInfoFile(beanClassName,  props);
52.       }
53.       catch (IOException e)
54.       {
55.         e.printStackTrace();
56.       }
57.     }
58.     return true;
59.   }
60.
61.   /**
62.    * Записывает исходный файл для класса BeanInfo.
```

```
63.    * @param beanClassName Имя класса bean-компонентов
64.    * @param props Таблица имен свойств и их аннотаций
65.    */
66.     private void writeBeanInfoFile(String beanClassName, Map<String, Property> props)
67.       throws IOException
68.     {
69.       JavaFileObject sourceFile = processingEnv.getFiler().createSourceFile(
70.         beanClassName + "BeanInfo");
71.       PrintWriter out = new PrintWriter(sourceFile.openWriter());
72.       int i = beanClassName.lastIndexOf(".");
73.       if (i > 0)
74.       {
75.         out.print("package ");
76.         out.print(beanClassName.substring(0, i));
77.         out.println(";");
78.       }
79.       out.print("public class ");
80.       out.print(beanClassName.substring(i + 1));
81.       out.println("BeanInfo extends java.beans.SimpleBeanInfo");
82.       out.println("{");
83.       out.println(" public java.beans.PropertyDescriptor[] getPropertyDescriptors()");
84.       out.println(" {");
85.       out.println(" try");
86.       out.println(" {");
87.       for (Map.Entry<String, Property> e : props.entrySet())
88.       {
89.         out.print(" java.beans.PropertyDescriptor ");
90.         out.print(e.getKey());
91.         out.println("Descriptor");
92.         out.print(" = new java.beans.PropertyDescriptor(\"");
93.         out.print(e.getKey());
94.         out.print("\", ");
95.         out.print(beanClassName);
96.         out.println(".class);");
97.         String ed = e.getValue().editor().toString();
98.         if (!ed.equals(""))
99.         {
100.           out.print(" ");
101.           out.print(e.getKey());
102.           out.print("Descriptor.setPropertyEditorClass(");
103.           out.print(ed);
104.           out.println(".class);");
105.        }
106.      }
107.      out.println(" return new java.beans.PropertyDescriptor[]");
108.      out.print(" {");
109.      boolean first = true;
110.      for (String p : props.keySet())
111.      {
112.        if (first) first = false;
113.        else out.print(",");
114.        out.println();
115.        out.print(" ");
116.        out.print(p);
117.        out.print("Descriptor");
118.      }
119.      out.println();
120.      out.println(" };");
```

```
121.      out.println(" }");
122.      out.println(" catch (java.beans.IntrospectionException e)");
123.      out.println(" {");
124.      out.println(" e.printStackTrace();");
125.      out.println(" return null;");
126.      out.println(" }");
127.      out.println(" }");
128.      out.println("}");
129.      out.close();
130.   }
131. }
```

Разработка байт-кода

Как аннотации могут обрабатываться во время выполнения и на уровне исходного кода, мы уже показали. Однако существует еще и третий вариант: аннотации еще также могут обрабатываться и на уровне байт-кода. Если аннотации не удаляются на уровне исходного кода, они присутствуют в файлах классов. Формат этих файлов документирован (`http://java.sun.com/docs/books/vmspec`). Данный формат является довольно сложным, и потому без специальных библиотек файлы классов было бы обрабатывать совсем не легко. Среди таких библиотек есть одна библиотека, которая называется BCEL (Bytecode Engineering Library — библиотека разработки байт-кода) и доступна по адресу `http://jakarta.apache.org/bcel`.

В этом разделе демонстрируется, как эту библиотеку можно использовать для добавления к аннотированным методам регистрационных сообщений. В случае снабжения метода следующей аннотацией:

```
@LogEntry(logger=имя_регистратора)
```

в начале этого метода добавляются байт-коды для следующего оператора:

```
Logger.getLogger(имя_регистратора).entering(имя_класса, имя_метода);
```

Например, если снабдить такой аннотацией метод `hashCode` класса `Item`:

```
@LogEntry(logger="global") public int hashCode()
```

тогда при каждом вызове этого метода будет выводиться приблизительно такое сообщение:

```
Aug 17, 2004 9:32:59 PM Item hashCode
FINER: ENTRY
```

Вот, что нужно сделать, чтобы добиться такого эффекта.

1. Загрузить байт-коды в файл классов.
2. Определить местонахождение всех методов.
3. Для каждого метода выполнить проверку на предмет наличия у него аннотации `LogEntry`.
4. Если таковая присутствует, добавить байт-коды для следующих инструкций в начале метода:

```
ldc имя_регистратора
invokestatic java/util/logging/Logger.getLogger:(Ljava/lang/String;)Ljava/
util/logging/Logger;
ldc имя_класса
ldc имя_метода
invokevirtual java/util/logging/Logger.entering:(Ljava/lang/String;Ljava/
lang/String;)V
```

Вставка этих байт-кодов может показаться сложной задачей, но BCEL делает ее выполнение довольно простым. Подробно описывать процесс анализа и вставки байт-кодов мы не будем. Самое главное можно увидеть в листинге 11.13, где приведена программа, редактирующая файл классов и вставляющая регистрационный вызов в начале всех имеющих аннотацию `LogEntry` методов.



Тем, кого интересуют детали создания байт-кодов, рекомендуем ознакомиться с руководством по BCEL, которое доступно по следующему адресу: `http://jakarta.apache.org/bcel/manual.html`.

Для компиляции и запуска программы `EntryLogger` необходима версия BCEL 5.3 или выше. (На момент написания настоящей главы версия 5.3 все еще находилась на этапе разработки. Если она до сих пор не готова, следует использовать промежуточную версию из репозитория Subversion.) Например, вот как добавляются инструкции в показанный в листинге 11.14 файл `Item.java`:

```
javac Item.java
javac -classpath .:bcel-версия.jar EntryLogger.java
java -classpath .:bcel-версия.jar EntryLogger Item
```

Попробуйте выполнить команду

```
javap -c Item
```

до и после изменения файла класса `Item`, и вы сможете увидеть вставленные инструкции в начале методов `hashCode`, `equals` и `compareTo`.

```
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`, код которой приведен в листинге 11.15, вставляет объекты `Item` в хеш-набор. Если вы запустите ее с измененным файлом классов, вы увидите следующие регистрационные сообщения:

```
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
[[descripion=Toaster, partNumber=1729], [descripion=Microwave, partNumber=4104]]
```

Обратите внимание на вызов метода `equals` при выполнении вставки одного и того же элемента дважды.

Этот пример иллюстрирует мощь подхода с применением байт-кодов. Аннотации используются для добавления в программу директив, а инструмент редактирования байт-кодов собирает эти директивы и соответствующим образом модифицирует инструкции виртуальной машины.

Листинг 11.13. Содержимое файла `EntryLogger.java`

```
1.  import java.io.*;
2.  import org.apache.bcel.*;
3.  import org.apache.bcel.classfile.*;
4.  import org.apache.bcel.generic.*;
5.  /**
6.   * Добавляет журналы для записей ко всем методам класса,
7.   * у которых имеется аннотация LogEntry.
8.   * @version 1.10 2007-10-27
9.   * @author Cay Horstmann
10.  */
11. public class EntryLogger
12. {
13.   /**
14.    * Добавляет код регистрации записей к заданному классу.
15.    * @param args Имя подлежащего исправлению класса
16.    */
17.   public static void main(String[] args)
18.   {
19.     try
20.     {
21.       if (args.length == 0) System.out.println("USAGE: java EntryLogger classname");
22.        else
23.        {
24.          JavaClass jc = Repository.lookupClass(args[0]);
25.          ClassGen cg = new ClassGen(jc);
26.          EntryLogger el = new EntryLogger(cg);
27.          el.convert();
28.          File f = new File(Repository.lookupClassFile(cg.getClassName()).getPath());
29.          cg.getJavaClass().dump(f.getPath());
30.        }
31.       }
32.       catch (Exception e)
33.       {
34.         e.printStackTrace();
35.       }
36.    }
37.
38.    /**
39.     * Создает инструмент EntryLogger, который вставляет
40.     * регистрационный код  в аннотированные методы
41.     * заданного класса.
42.     * @param cg Класс
43.     */
44.   public EntryLogger(ClassGen cg)
45.   {
46.     this.cg = cg;
47.     cpg = cg.getConstantPool();
48.   }
```

```
49.   /**
50.    * Преобразует класс путем вставки регистрационных вызовов.
51.    */
52.   public void convert() throws IOException
53.   {
54.    for (Method m : cg.getMethods())
55.    {
56.     AnnotationEntry[] annotations = m.getAnnotationEntries();
57.     for (AnnotationEntry a : annotations)
58.     {
59.      if (a.getAnnotationType().equals("LLogEntry;"))
60.      {
61.       for (ElementValuePair p : a.getElementValuePairs())
62.       {
63.        if (p.getNameString().equals("logger"))
64.        {
65.          String loggerName = p.getValue().stringifyValue();
66.           cg.replaceMethod(m, insertLogEntry(m, loggerName));
67.        }
68.       }
69.      }
70.     }
71.    }
72.   }
73.   /**
74.    * Добавляет вызов регистратора в начало метода.
75.    * @param m метод
76.    * @param loggerName Имя подлежащего вызову регистратора
77.    */
78.  private Method insertLogEntry(Method m, String loggerName)
79.  {
80.   MethodGen mg = new MethodGen(m, cg.getClassName(), cpg);
81.   String className = cg.getClassName();
82.   String methodName = mg.getMethod().getName();
83.   System.out.printf("Adding logging instructions to %s.%s%n", className, methodName);
84.
85.    int getLoggerIndex = cpg.addMethodref("java.util.logging.Logger", "getLogger",
86.        "(Ljava/lang/String;)Ljava/util/logging/Logger;");
87.    int enteringIndex = cpg.addMethodref("java.util.logging.Logger", "enter ing",
88.       "(Ljava/lang/String;Ljava/lang/String;)V");
89.
90.    InstructionList il = mg.getInstructionList();
91.    InstructionList patch = new InstructionList();
92.    patch.append(new PUSH(cpg, loggerName));
93.    patch.append(new INVOKESTATIC(getLoggerIndex));
94.    patch.append(new PUSH(cpg, className));
95.    patch.append(new PUSH(cpg, methodName));
96.    patch.append(new INVOKEVIRTUAL(enteringIndex));
97.    InstructionHandle[] ihs = il.getInstructionHandles();
98.    il.insert(ihs[0], patch);
99.
100.    mg.setMaxStack();
101.    return mg.getMethod();
102.  }
103.  private ClassGen cg;
104.  private ConstantPoolGen cpg;
105. }
```

Листинг 11.14. Содержимое файла `Item.java`

```
1.  /**
2.   * Элемент с описанием и номером части.
3.   * @version 1.00 2004-08-17
4.   * @author Cay Horstmann
5.   */
6.  public class Item
7.  {
8.   /**
9.    * Создает элемент.
10.   * @param aDescription Описание элемента
11.   * @param aPartNumber Номер части элемента
12.   */
13.  public Item(String aDescription, int aPartNumber)
14.  {
15.   description = aDescription;
16.   partNumber = aPartNumber;
17.  }
18.
19.  /**
20.   * Получает описание данного элемента.
21.   * @return Описание
22.   */
23.  public String getDescription()
24.  {
25.     return description;
26.  }
27.
28.  public String toString()
29.  {
30.    return "[description=" + description + ", partNumber=" + partNumber + "]";
31.  }
32.
33.  @LogEntry(logger = "global")
34.  public boolean equals(Object otherObject)
35.  {
36.    if (this == otherObject) return true;
37.    if (otherObject == null) return false;
38.    if (getClass() != otherObject.getClass()) return false;
39.    Item other = (Item) otherObject;
40.    return description.equals(other.description) && partNumber == other.partNumber;
41.  }
42.
43.  @LogEntry(logger = "global")
44.  public int hashCode()
45.  {
46.    return 13 * description.hashCode() + 17 * partNumber;
47.  }
48.
49.  private String description;
50.  private int partNumber;
51. }
```

Листинг 11.15. Содержимое файла `SetTest.java`

```
1.  import java.util.*;
2.  import java.util.logging.*;
3.
4.  /**
5.   * @version 1.01 2007-10-27
6.   * @author Cay Horstmann
7.   */
8.  public class SetTest
9.  {
10.   public static void main(String[] args)
11.   {
12.     Logger.getLogger(Logger.GLOBAL_LOGGER_NAME).setLevel(Level.FINEST);
13.     Handler handler = new ConsoleHandler();
14.     handler.setLevel(Level.FINEST);
15.     Logger.getLogger(Logger.GLOBAL_LOGGER_NAME).addHandler(handler);
16.
17.     Set<Item> parts = new HashSet<Item>();
18.     parts.add(new Item("Toaster", 1279));
19.     parts.add(new Item("Microwave", 4104));
20.     parts.add(new Item("Toaster", 1279));
21.     System.out.println(parts);
22.   }
23. }
```

Изменение байт-кодов во время загрузки

В предыдущем разделе был показан инструмент, умеющий редактировать файлы классов. Однако добавлять еще один такой инструмент в процесс сборки может оказаться обременительным. Поэтому существует довольно привлекательный альтернативный вариант и заключается он в откладывании создания байт-кодов до *этапа загрузки*, на котором загрузчик классов загружает классы.

До появления версии Java SE 5.0 для реализации подобного подхода требовалось разрабатывать специальный загрузчик классов. Сегодня в *инструментальном API* (instrumentation API) имеется перехватчик, позволяющий устанавливать преобразователь байт-кодов. Этот преобразователь должен устанавливаться перед вызовом главного (`main`) метода программы. Удовлетворяется данное требование путем определения *агента* (то есть библиотеки, загружаемой для осуществления каким-нибудь образом мониторинга за программой). Код этого агента может выполнять необходимые ему операции по инициализации в методе `premain`.

Требуемые для создания агента шаги выглядят следующим образом.

1. Реализовать класс с помощью метода

   ```
   public static void premain(String arg, Instrumentation instr)
   ```

   Этот метод вызывается при загрузке агента. Агент может получать один единственный аргумент командной строки, передаваемый в параметре `arg`. Параметр `instr` может использоваться для установки различных перехватчиков.

2. Создать файл манифеста, устанавливающий атрибут `Premain-Class`, например:

   ```
   Premain-Class: EntryLoggingAgent
   ```

3. Упаковать код агента и манифест в файл JAR, например:

   ```
   javac -classpath .:bcel-версия.jar EntryLoggingAgent
   jar cvfm EntryLoggingAgent.jar EntryLoggingAgent.mf Entry*.class
   ```

Чтобы запустить программу Java вместе с агентом, нужно использовать следующие опции командной строки:

```
java -javaagent:AgentJARFile=agentArgument . . .
```

Например, чтобы запустить программу `SetTest` с агентом регистрации записей, потребуется выполнить следующую команду:

```
javac SetTest.java
java -javaagent:EntryLoggingAgent.jar=Item -classpath .:bcel-версия.jar SetTest
```

Аргумент `Item` — это имя класса, который агенту следует изменить.

В листинге 11.16 приведен код необходимого агента. Этот агент устанавливает преобразователь файлов классов. Этот преобразователь сначала проверяет, соответствует ли имя класса аргументу агента. Если да, тогда он использует приведенный в предыдущем разделе класс `EntryLogger` для изменения байт-кодов. В файл, однако, измененные байт-коды не сохраняются. Вместо этого преобразователь возвращает их для загрузки в виртуальную машину (рис. 11.4). Другими словами, получается, что при таком подходе осуществляется "оперативное" (just in time) изменение байт-кодов.

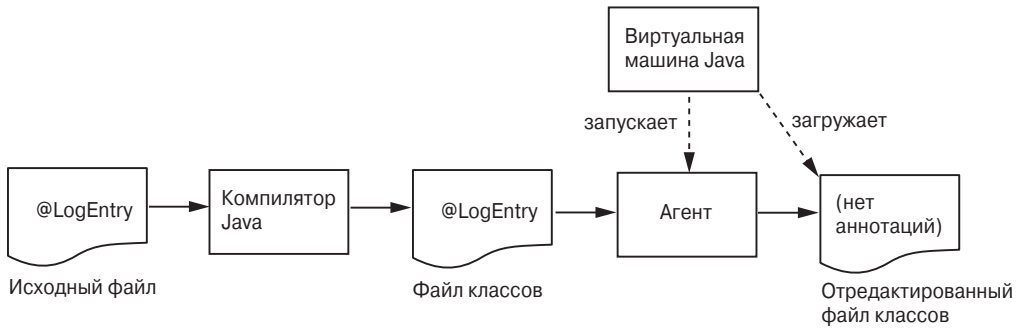


Рис. 11.4. Изменение классов во время загрузки

Листинг 11.16. Содержимое файла `EntryLoggingAgent.java`

```
1.  import java.lang.instrument.*;
2.  import java.io.*;
3.  import java.security.*;
4.  import org.apache.bcel.classfile.*;
5.  import org.apache.bcel.generic.*;
6.
7.  /**
8.   * @version 1.00 2004-08-17
9.   * @author Cay Horstmann
10.  */
11. public class EntryLoggingAgent
12. {
13.   public static void premain(final String arg, Instrumentation instr)
14.   {
15.     instr.addTransformer(new ClassFileTransformer()
16.     {
17.       public byte[] transform(ClassLoader loader, String className,
18.           Class<?> cl, ProtectionDomain pd, byte[] data)
19.       {
20.         if (!className.equals(arg)) return null;
```

```
21.         try
22.         {
23.           ClassParser parser = new ClassParser(new ByteArrayInputStream(data),
24.                                 className + ".java");
25.           JavaClass jc = parser.parse();
26.           ClassGen cg = new ClassGen(jc);
27.           EntryLogger el = new EntryLogger(cg);
28.           el.convert();
29.           return cg.getJavaClass().getBytes();
30.         }
31.         catch (Exception e)
32.         {
33.           e.printStackTrace();
34.           return null;
35.         }
36.       }
37.      });
38.    }
39. }
```

В этой главе было показано, как можно делать следующие вещи:

- добавлять аннотации в Java-программы;
- создавать свои собственные интерфейсы аннотаций;
- реализовать инструменты, применяющие аннотации.

Кроме того, здесь также были продемонстрированы три методики для обработки кода — написание сценариев, компиляция программ Java и обработка аннотаций. Первые две из них являются довольно простыми. Что касается создания инструментов обработки аннотаций, то эта методика, бесспорно, является сложной и точно не такой, которой рискнут пользоваться многие разработчики. В этой главе были приведены лишь базовые сведения, которые могут пригодиться для понимания внутренних особенностей встречающихся инструментов обработки аннотаций и, возможно, вызвать интерес к созданию своих собственных.

Последняя глава этой книги посвящена рассмотрению API-интерфейса программирования платформенно-ориентированных методов, который позволяет смешивать код Java и код C/C++.

Глава 12

ПЛАТФОРМЕННО-ОРИЕНТИРОВАННЫЕ МЕТОДЫ

В ЭТОЙ ГЛАВЕ...

- Вызов функции на языке C из Java-программы
- Числовые параметры и возвращаемые значения
- Строковые параметры
- Получение доступа к полям объектов
- Кодирование сигнатуры
- Вызов Java-методов
- Доступ к элементам массивов
- Обработка ошибок
- API вызова
- Пример обращения к реестру Windows

Хотя подход с применением исключительно языка Java в принципе является замечательным, бывают ситуации, когда просто необходимо создавать (или использовать) код, написанный на каком-то другом языке. (Такой код обычно называют платформенно-ориентированным.)

В частности, когда язык Java только появился, многие люди считали, что для ускорения работы критических частей Java-приложения, будет выгоднее использовать язык C или C++. На практике, однако, такой подход редко оказывался эффективным. Проведенная на конференции по JavaOne в 1996 г. презентация очень четко показала это. Создатели криптографической библиотеки из компании Sun Microsystems продемонстрировали, что скорость выполнения криптографических функций только на основе Java-кода была вполне удовлетворительной. Конечно, она уступала скорости выполнения тех же функций на языке C, но отличие было не очень существенным. Дело в том, что реализация платформы Java была значительно быстрее, чем сетевой ввод-вывод. Именно это и оказалось настоящей причиной более низкой производительности.

Разумеется, у подхода с применением платформенно-ориентированного кода имеются свои недостатки. В случае написания части приложения на каком-то другом языке требуется обязательно предоставлять отдельную специализированную библиотеку для каждой платформы, которую планируется поддерживать. Код, который пишется на языке C или C++, не предусматривает защиты против некорректного обращения к памяти из-за неправильно заданного указателя. Это значит, что можно очень легко ненамеренно написать такие платформенно-ориентированные методы, которые будут нарушать работу программы или заражать операционную систему.

Мы рекомендуем использовать платформенно-ориентированный код только в тех случаях, когда это действительно необходимо. В частности, существуют три случая, когда платформенно-ориентированный код может оказаться правильным выбором:

- когда приложению требуется доступ к таким системным функциям или устройствам, получение доступа к которым посредством платформы Java невозможно;
- когда имеется приличное количество проверенного или отлаженного кода на другом языке и известны способы его переноса на все желаемые целевые платформы;
- когда посредством проведения сопоставительных испытаний было обнаружено, что код на Java будет выполняться гораздо медленнее, чем его эквивалент на другом языке.

В состав платформы Java входит специальный API-интерфейс для обеспечения взаимодействия с платформенно-ориентированным кодом на C, который называется JNI (Java Native Interface — платформенно-ориентированный интерфейс Java).



Для написания платформенно-ориентированных методов вместо C может еще также использоваться и язык C++. Он имеет ряд преимуществ, например, обеспечивает более строгий контроль соответствия типов и более удобный доступ к JNI-функциям. Однако JNI не поддерживает никакого отображения между классами Java и классами C++.

Вызов функции на языке C из Java-программы

Предположим, что имеется некая функция на языке C, которая выполняет определенную полезную операцию и которую по той или иной причине не хочется реализовать заново на Java. Дабы упростить иллюстрацию, давайте для начала представим, что это простая C-функция, которая выводит приветственное сообщение.

В языке программирования Java для организации вызова платформенно-ориентированного метода нужно применять ключевое слово `native`, а сам метод, конечно же, помещать в класс. Как именно это делается можно увидеть в листинге 12.1.

Ключевое слово `native` предупреждает компилятор о том, что метод будет определяться внешним образом. Естественно, платформенно-ориентированные методы не будут содержать код на языке Java, поэтому объявление метода состоит только из одного заголовка, за которым сразу же следует замыкающая точка с запятой. Это делает объявления платформенно-ориентированных методов похожими на объявления абстрактных методов.

Листинг 12.1. Содержимое файла `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.  }
```

Конкретно в данном примере платформенно-ориентированный метод еще также объявляется и как статический (`static`). Платформенно-ориентированные методы могут быть как статическими, так и не статическими. Мы начали со статического метода, поскольку пока не хотим иметь дела с передачей параметров.

Фактически этот класс уже можно скомпилировать, но при использовании его в программе виртуальная машина Java уведомит о том, что не знает, как искать функцию `greeting`, выдав сообщение `UnsatisfiedLinkError`. Чтобы устранить эту проблему, нужно реализовать платформенно-ориентированный код, то есть написать соответствующую функцию на C. Имя этой функции должно обязательно *точно* соответствовать ожиданиям виртуальной машины Java. Чтобы это было так, нужно соблюсти следующие правила.

1. Использовать полное имя метода из Java, то есть `HelloNative.greeting`. Если класс находится в пакете, тогда добавить перед именем метода еще и имя этого пакета: `com.horstmann.HelloNative.greeting`.
2. Заменить все точки символами подчеркивания и присоединить префикс `Java_`, то есть, например, сделать так: `Java_HelloNative_greeting` или `Java_com_horstmann_HelloNative_greeting`.
3. Если в имени класса присутствуют символы, не являющиеся ни буквами, ни цифрами формата ASCII (например, _ или $), или символы формата Unicode с кодом, превышающим `\u007F`, заменить их последовательностью `_0`*xxxx*, указав на месте *xxxx* четырехзначное шестнадцатеричное Unicode-значение заменяемого символа.



В случае *перегрузки* платформенно-ориентированных методов, то есть в случае предоставления нескольких платформенно-ориентированных методов с одинаковым именем, в конце имени каждого из них нужно присоединять два символа подчеркивания со следующими далее закодированными соответствующим образом типами аргументов. (О том, как кодируются аргументы, более подробно будет рассказываться позже в этой главе.) Например, в случае предоставления одного платформенно-ориентированного метода с именем `greeting()` и еще одного платформенно-ориентированного метода с именем `greeting(int repeat)`, имя первого должно выглядеть как `Java_HelloNative_greeting__`, а имя второго — как `Java_HelloNative_greeting__I`.

Разумеется, вручную этого никто не делает, поскольку существует специальная утилита javah, умеющая автоматически генерировать имена функций правильным образом. Чтобы воспользоваться этой утилитой, сначала нужно скомпилировать исходный файл, приведенный в листинге 12.1:

```
javac HelloNative.java
```

После этого уже можно вызывать утилиту, чтобы она сгенерировала из скомпилированного файла класса заголовочный файл на языке C. Исполняемый файл утилиты javah находится в каталоге *jdk*/bin. Вызывать ее нужно с использованием имени класса, то есть точно так же как и компилятор Java. Например:

```
javah HelloNative
```

Выполнение этой команды приведет к созданию заголовочного файла HelloNative.h, содержимое которого приведено в листинге. 12.2.

Листинг 12.2. Содержимое файла `HelloNative.h`

```
1.  /* DO NOT EDIT THIS FILE - it is machine generated */
2.  #include <jni.h>
3.  /* Header for class HelloNative */
4.
5.  #ifndef _Included_HelloNative
6.  #define _Included_HelloNative
7.  #ifdef __cplusplus
8.  extern "C" {
9.  #endif
10. /*
11.  * Class:     HelloNative
12.  * Method:    greeting
13.  * Signature: ()V
14.  */
15. JNIEXPORT void JNICALL Java_HelloNative_greeting
16.    (JNIEnv *, jclass);
17.
18. #ifdef __cplusplus
19. }
20. #endif
21. #endif
```

В листинге видно, что данный файл содержит объявление функции Java_HelloNative_greeting. (Определения макросов JNIEXPORT и JNICALL находятся в заголовочном файле jni.h. Они представляют зависящие от компилятора спецификаторы для экспортируемых функций, которые поступают из динамически загружаемой библиотеки.)

Теперь осталось просто скопировать прототип функции из заголовочного файла в файл исходного кода и написать для этой функции код реализации, как показано в листинге 12.3.

Листинг 12.3. Содержимое файла `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.  JNIEXPORT void JNICALL Java_HelloNative_greeting(JNIEnv* env, jclass cl)
9.  {
10.   printf("Hello Native World!\n");
11. }
```

На аргументы `env` и `cl` в этой простой функции пока не нужно обращать внимания. Об их использовании более подробно будет рассказываться позже в этой главе.



Для реализации платформенно-ориентированных методов можно использовать и язык C++. Однако в таком случае функции, реализующие эти платформенно-ориентированные методы, нужно обязательно объявлять как `extern "C"`. (Это не позволит компилятору C++ "коверкать" имена методов.) Например:

```
extern "C"
JNIEXPORT void JNICALL Java_HelloNative_greeting(JNIEnv* env, jclass cl)
{
    cout << "Hello, Native World!" << endl;
}
```

Далее платформенно-ориентированный код на C потребуется скомпилировать в динамически загружаемую библиотеку. Детали зависят от применяемого компилятора.

Например, для C-компилятора GNU в операционной системе Linux, необходимая команда будет выглядеть так:

```
gcc -fPIC -I jdk/include -I jdk/include/linux -shared
    -o libHelloNative.so HelloNative.c
```

Для компилятора Sun в ОС Solaris необходимая команда будет выглядеть так:

```
cc -G -I jdk/include -I jdk/include/solaris
   -o libHelloNative.so HelloNative.c
```

А для компилятора Microsoft C++ в Windows она будет выглядеть так:

```
cl -I jdk\include -I jdk\include\win32 -LD HelloNative.c
   -FeHelloNative.dll
```

Во всех этих командах *jdk* — это каталог, в котором содержится пакет JDK.



В случае использования компилятора Microsoft из командной оболочки сначала необходимо запустить файл `vcvars32.bat` или `vsvars32.bat`. Этот командный файл установит путь и переменные окружения, необходимые компилятору. Найти его можно либо в каталоге `c:\Program Files\Microsoft Visual Studio .NET 2003\Common7\tools`, либо в каталоге `c:\Program Files\Microsoft Visual Studio 8\VC`, либо в каком-то другом каталоге с подобным названием.

Кроме того, еще также можно использовать и бесплатно распространяемую среду программирования Cygwin, доступную для загрузки на сайте `http://www.cygwin.com`. В ее состав входит компилятор GNU C и библиотеки для написания кода в стиле Unix под управлением операционной системы Windows. В случае использования Cygwin необходимая команда будет выглядеть следующим образом:

```
gcc -mno-cygwin -D __int64="long long" -I jdk/include/
    -I jdk/include/win32
    -shared -Wl,--add-stdcall-alias -o HelloNative.dll HelloNative.c
```

Всю эту команду следует вводить в одной строке.



Windows-версия заголовочного файла `jni_md.h` будет содержать следующее характерное для компилятора Microsoft объявление типа:

```
typedef __int64 jlong;
```

Из-за этого, в случае применения компилятора GNU, может возникнуть желание отредактировать данный файл, например, таким образом:

```
#ifdef __GNUC__
    typedef long long jlong;
#else
    typedef __int64 jlong;
#endif
```

В качестве альтернативного варианта можно указывать при вызове компилятора опцию `-D __int64="long long"`.

И, наконец, последнее, что нужно сделать — это добавить в программу вызов метода `System.loadLibrary()`. Для уверенности в том, что виртуальная машина будет загружать библиотеку перед первым использованием класса, следует использовать статический блок инициализации, как показано в листинге 12.4. На рис. 12.1 приведена общая схема обработки платформенно-ориентированного кода.

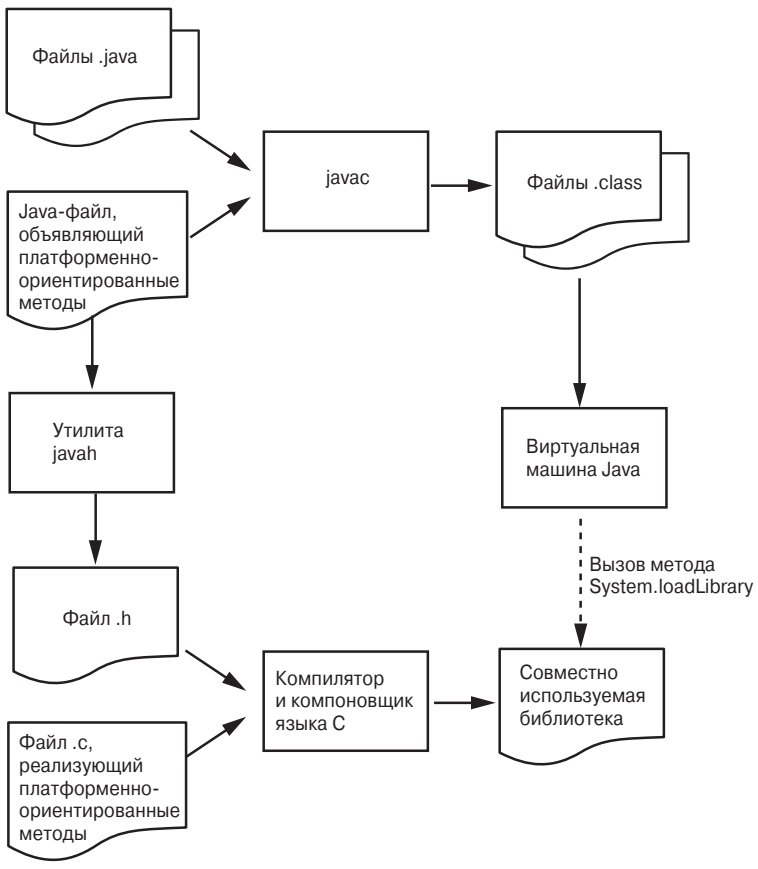


Рис. 12.1. Обработка платформенно-ориентированного кода

Листинг 12.4. Содержимое файла `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. }
```

После компиляции и запуска этой программы на экране появится сообщение `Hello, Native World!`.



Тем, кто работает в Linux, потребуется включить в путь к библиотеке текущий каталог. Это можно сделать либо путем установки переменной окружения `LD_LIBRARY_PATH`:

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

либо путем установки системного свойства `java.library.path`:

```
java -Djava.library.path=. HelloNativeTest
```

Конечно, сам по себе результат не особенно впечатляет. Однако если вспомнить, что данное сообщение генерируется командой, написанной на языке C, а не на языке Java, то сразу становится видно, что нам уже удалось сделать первые шаги на пути по устранению разрыва между этими двумя языками!

В общем необходимые для связывания платформенно-ориентированного метода с Java-программой действия сводятся к следующему.

1. Объявить платформенно-ориентированный метод в Java-классе.
2. Запустить утилиту `javah` для получения заголовочного файла с объявлением метода на языке C.
3. Реализовать платформенно-ориентированный метод на языке C.
4. Поместить полученный код в совместно используемую библиотеку.
5. Загрузить библиотеку в Java-программу.



`java.lang.System 1.0`

- `void loadLibrary(String libname)`

  Загружает библиотеку с заданным именем. Библиотека находится в том каталоге, который указан в пути поиска библиотеки. Конкретный способ поиска библиотеки зависит от используемой операционной системы.



Некоторым совместно используемым библиотекам с платформенно-ориентированным кодом требуется выполнять код инициализации. Любой код инициализации можно размещать в методе `JNI_OnLoad`. Тогда при завершении работы виртуальная машина Java будет аналогичным образом вызывать еще и метод `JNI_OnUnload`. Прототипы этих методов выглядят так, как показано ниже:

```
jint JNI_OnLoad(JavaVM* vm, void* reserved);
void JNI_OnUnload(JavaVM* vm, void* reserved);
```

Метод `JNI_OnLoad` должен возвращать самую раннюю версию виртуальной машины, которая необходима для работы библиотеки, например, `JNI_VERSION_1_2`.

Числовые параметры и возвращаемые значения

При передаче числовых параметров между языками C и Java, нужно обязательно знать о том, какие типы в них являются соответствующими. Например, в языке C есть типы данных `int` и `long`, но их реализация зависит от платформы. На одних платформах тип данных `int` представляет 16-битовое значение, а на других — 32-битовое. В языке Java, однако, тип данных `int` *всегда* представляет только 32-битовое целочисленное значение. По этой причине в JNI и предлагаются такие типы, как `jint`, `jlong` и т.д.

В табл. 12.1 приведена схема соответствий между типами в языках Java и C.

Таблица 12.1. Типы данных в языках Java и C

Язык Java	Язык 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

В заголовочном файле `jni.h` эти типы данных объявляются с помощью операторов `typedef` в качестве эквивалентных типов на целевой платформе. Помимо этого в этом файле еще также определяются и константы `JNI_FALSE = 0` и `JNI_TRUE = 1`.

Использование функции `printf()` для форматирования чисел

До появления версии Java SE 5.0 в языке Java не было никакого непосредственного аналога C-функции `printf()`. В следующих примерах предполагается, что по той или иной причине в распоряжении по-прежнему имеется только старая версия JDK и потому принимается решение реализовать точно такую же функциональность за счет вызова C-функции `printf()` в платформенно-ориентированном методе.

В листинге 12.5 показан класса `Printf1`, который использует платформенно-ориентированный метод для отображения с указанной точностью числа с плавающей точкой в поле заданной длины.

Листинг 12.5. Содержимое файла `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. }
```

Обратите внимание, что при реализации метода на языке C все параметры типа `int` и `double` меняются на параметры типа `jint` и `jdouble`, как показано в листинге 12.6.

Листинг 12.6. Содержимое файла `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. }
```

Эта функция просто компонует форматирующую строку "`%w.pf`" с помощью переменной `fmt` и затем вызывает `printf()`, после чего возвращает количество выведенных символов.

В листинге 12.7 приведена тестовая программа, которая демонстрирует применение класса `Printf1`.

Листинг 12.7. Содержимое файла `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

Теперь давайте посмотрим, как можно передавать строки в и из платформенно-ориентированных методов. Как известно, в языке Java строки представляют собой последовательности кодовых точек в формате UTF-16, а в языке C — оканчивающиеся нулевым символом последовательности байтов, то есть способы представления строк в этих двух языках существенно отличаются. В JNI предлагается два набора функций для манипулирования строками: первый позволяет преобразовывать Java-строки в байтовые последовательности в "модифицированном формате UTF-8", а второй — в массивы значений в формате UTF-16, то есть в массивы `jchar`. (О формате UTF-8, "модифицированном UTF-8" и UTF-16 более подробно рассказывалось в главе 12 первого тома. Напомним, что в кодировке "модифицированного формата UTF-8" символы ASCII оставляются без изменений, но зато все остальные Unicode-символы кодируются в многобайтовые последовательности.)



Стандартный и модифицированный форматы UTF-8 различаются только дополнительными символами с кодами, превышающими `0xFFFF`. В стандартной кодировке UTF-8 эти символы кодируются 4-байтовыми последовательностями. Согласно модифицированному формату, символ сначала представляется парой "заменителей" в кодировке UTF-16, а затем каждый заменитель кодируется средствами UTF-8. В результате получается последовательность из 6 байт. Такое решение нельзя назвать изящным, но другого выхода нет — спецификация виртуальной машины Java была написана тогда, когда длина Unicode-символа ограничивалась 16 битами.

Если в имеющемся коде на языке C уже применяется Unicode, тогда эффективнее всего будет использовать второй набор функций преобразования строк. Если же все строки содержат только символы ASCII, тогда можно использовать и просто функции преобразования строк в модифицированный формат UTF-8.

Платформенно-ориентированный метод со строковым параметром типа `String` фактически получает значение типа `jstring`. Платформенно-ориентированный метод с возвращаемым значением типа `String` должен обязательно 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 либо символы в формате ASCII, либо, что бывает гораздо чаще, байтовые последовательности в кодировке "модифицированного формата 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;
}
```



В случае отсутствия соответствующих указаний в этой главе везде подразумевается, что речь идет о коде на языке C.

Во всех вызовах функций JNI используется указатель `env`, который является первым параметром каждого платформенно-ориентированного метода. Он ссылается на таблицу с указателями функций (рис. 12.2). Из-за этого перед каждым вызовом функции JNI нужно обязательно добавлять префикс `(*env)->` для разыменования фактического указателя функции. Более того, указатель `env` еще также является и первым параметром каждой JNI-функции.



В языке C++ осуществлять доступ к JNI-функциям немного проще. В версии C++ класса `JNIEnv` имеются функции-члены, которые умеют автоматически выполнять поиск указателей функций. Например, вызвать функцию `NewStringUTF` в языке C++ можно следующим образом:

```
jstr = env->NewStringUTF(greeting);
```

Обратите внимание на то, что указателя `JNIEnv` в списке параметров вызова нет.

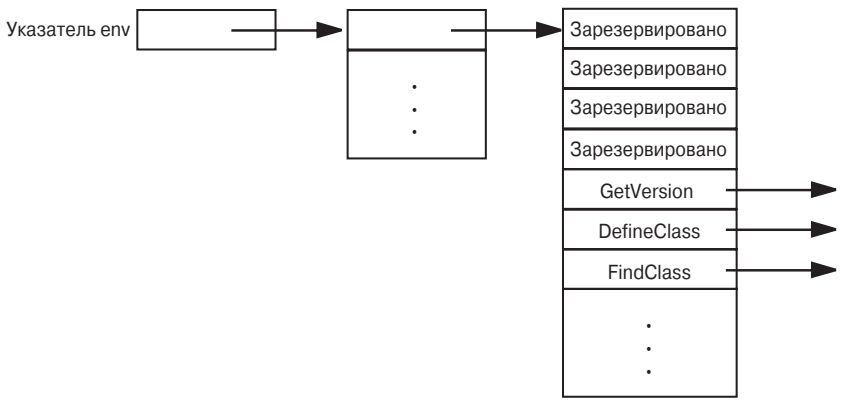


Рис. 12.2. Указатель `env`

Функция `NewStringUTF()` позволяет создавать новый объект `jstring`. Для считывания содержимого уже существующего объекта `jstring` применяется функция `GetStringUTFChars()`. Эта функция возвращает указатель `const jbyte*`, указывающий на описывающие строку символы в модифицированном формате UTF-8. Учтите, что некоторые виртуальные машины Java могут использовать данный формат для внутреннего представления строки, из-за чего возможно получение указателя и на реальную Java-строку. Поскольку Java-строки считаются постоянными, *очень важно* серьезно относится к обозначению `const` и не пытаться ничего записывать в такой символьный массив. С другой стороны, если виртуальная машина применяет для внутреннего представления строк формат UTF-16 или UTF-32, то вызов функции `GetStringUTFChars()` будет приводить к выделению нового блока памяти и его заполнению эквивалентными символами в модифицированном формате UTF-8.

После завершения работы со строкой следует с помощью функции `ReleaseStringUTFChars()` сообщать об этом виртуальной машине для того, чтобы она могла освободить занимаемую данной строкой память. (Как известно, процесс "сборки мусора" происходит в отдельном потоке и может прерывать выполнение платформенно-ориентированных методов.) А это требует вызова функции `ReleaseStringUTFChars()`.

В качестве альтернативного варианта еще также можно и предоставлять свой собственный буфер для хранения символов строки путем вызова методов `GetStringRegion()` или `GetStringUTFRegion()`.

И, наконец, функция `GetStringUTFLength()` возвращает количество символов, необходимых для представления строки в модифицированном формате UTF-8.



Описание API-интерфейса JNI можно найти по следующему адресу:
`http://java.sun.com/javase/6/docs/technotes/guides/jni/index.html`.



Получение доступа к Java-строкам из кода на языке C

- `jstring NewStringUTF(JNIEnv* env, const char bytes[])`

  Возвращает новый Java-объект строки из оканчивающейся нулевым байтом последовательности байтов в модифицированном формате UTF-8, или значение `NULL`, если сконструировать строку не удается.

- `jsize GetStringUTFLength(JNIEnv* env, jstring string)`

  Возвращает количество байтов, требуемое для кодирования строки в модифицированный формат UTF-8 (без учета завершающего нулевого байта).

- `const jbyte* GetStringUTFChars(JNIEnv* env, jstring string, jboolean* isCopy)`

  Возвращает указатель на строку в модифицированном формате UTF-8 или значение `NULL`, если создать символьный массив в таком формате невозможно. Этот указатель остается действительным до тех пор, пока не будет вызвана функция `ReleaseStringUTFChars`. Параметр `isCopy` указывает на переменную `jboolean`, которая в случае создания копии заполняется значением `JNI_TRUE`, а в противном случае — значением `JNI_FALSE`.

- `void ReleaseStringUTFChars(JNIEnv* env, jstring string, const jbyte bytes[])`

  Сообщает виртуальной машине, что платформенно-ориентированному коду больше не требуется доступ к строке Java через `bytes` (указатель, возвращаемый функцией `GetStringUTFChars()`).

- `void GetStringRegion(JNIEnv *env, jstring string, jsize start, jsize length, jchar *buffer)`

  Копирует последовательность двойных байтов в формате UTF-16 из 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 строки в предоставляемый пользователем буфер. Емкости буфера должно хватать для умещения всех байтов. В худшем случае копируется только 3 × `length` байтов.

- `jstring NewString(JNIEnv* env, const jchar chars[], jsize length)`

  Возвращает новый объект строки Java из строки в формате Unicode или значение `NULL`, если создать строку Java не удается.

  Параметры:

`env`	Указатель JNI
`chars`	Строка в формате 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 формате Unicode или `NULL`, если создать символьный массив в таком формате не удается. Указатель действует до тех пор, пока не будет вызвана функция `ReleaseStringUTFChars()`. Параметр `isCopy` содержит либо значение `NULL`, либо указатель на переменную `jboolean`, которая в случае создания копии заполняется значением `JNI_TRUE`, а в противном случае — значением `JNI_FALSE`.

- `void ReleaseStringChars(JNIEnv* env, jstring string, const jchar chars[])`

  Информирует виртуальную машину о том, что платформенно-ориентированному коду уже больше не требуется доступ к строке Java через `chars` (указатель, возвращаемый функцией `GetStringChars`).

Давайте теперь испытаем все эти функции на практике и напишем класс, способный вызывать C-функцию `sprintf()`. Нам бы хотелось, чтобы эта функция вызывалась так, как показано в листинге 12.8.

Листинг 12.8. Содержимое файла `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 показан класс с платформенно-ориентированным методом `sprint()`.

Листинг 12.9. Содержимое файла `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, получается, что функция языка 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 языке C, приведен в листинге 12.10. Обратите внимание на то, что для считывания параметра форматирования `format` вызывается функция `GetStringUTFChars()`, для генерации возвращаемого значения — функция `NewStringUTF()`, а для уведомления виртуальной машины о том, что доступ к строке больше не требуется — функция `ReleaseStringUTFChars()`.

Листинг 12.10. Содержимое файла `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 Строка, содержащая спецификатор формата printf
13.  * (наподобие "%8.2f"). Подстроки "%%" пропускаются
14.  * @return Указатель на спецификатор формата (без '%')
15.  * или NULL, если уникального спецификатора формата не было.
16.  */
17. char* find_format(const char format[])
18. {
19.   char* p;
20.   char* q;
21.
22.     p = strchr(p + 2, '%');
23.   if (p == NULL) return NULL;
24.    /* Проверка % на предмет уникальности */
25.   p = strchr(format, '%');
26.   while (p != NULL && *(p + 1) == '%') /* пропуск %% */
27.   p++;
28.   q = strchr(p, '%');
29.   while (q != NULL && *(q + 1) == '%') /* пропуск %% */
30.     q = strchr(q + 2, '%');
31.
32.   if (q != NULL) return NULL; /* % не уникальный */
33.   q = p + strspn(p, " -0+#"); /* пропуск последних флагов */
34.   q += strspn(q, "0123456789");/* пропуск последней ширины поля */
```

```
35.    if (*q == '.') { q++; q += strspn(q, "0123456789"); }
36.      /* пропуск последнего значения точности */
37.    if (strchr("eEfFgG", *q) == NULL) return NULL;
38.      /* формат, отличный от формата 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 этой функции мы предпочли не усложнять механизм обработки ошибок. Если отвечающий за отображение числа с плавающей запятой код форматирования выглядит не как `%w.pc` (где `c` — один из символов `e`, `E`, `f`, `g`, `G`), тогда форматирование числа просто не выполняется. Как можно делать так, чтобы платформенно-ориентированный метод выдавал исключение, мы покажем чуть позже в этой главе.

Получение доступа к полям объектов

Все рассматривавшиеся пока что платформенно-ориентированные методы были статическими с числовыми и строковыми параметрами. Теперь речь пойдет уже о методах, способных работать с объектами произвольного типа. Для примера мы попробуем реализовать с помощью платформенно-ориентированного метода метод класса `Employee`, который уже приводился в главе 4 первого тома этой книги. Опять-таки, данный пример не является таким, который можно часто встретить в реальном мире, но зато он позволяет прекрасно проиллюстрировать способы получения доступа к полям объекта из платформенно-ориентированного метода.

Получение доступа к полям экземпляра

Чтобы увидеть, как можно получать доступ полям экземпляра из платформенно-ориентированного метода, давайте снова реализуем метод `raiseSalary()`. Код этого метода на языке Java выглядит так:

```
public native void raiseSalary(double byPercent)
{
   salary *= 1 + byPercent / 100;
}
```

Теперь превратим его в платформенно-ориентированный метод. В отличие от платформенно-ориентированных методов, которые приводились в предыдущих примерах, данный метод статическим не является. Поэтому запуск утилиты `javah` приводит к получению такого прототипа:

```
JNIEXPORT void JNICALL Java_Employee_raiseSalary(JNIEnv *,
  jobject, jdouble);
```

Обратите внимание на второй параметр, который теперь имеет тип `jobject`, а не `jclass`. Фактически он является эквивалентом ссылки `this`. Статические методы получают ссылку на класс, а нестатические — на конкретный объект.

Обеспечим доступ к полю `salary` неявного параметра. В Java 1.0 подобная задача решалась достаточно просто. Программист мог непосредственно получить доступ к полям данных объекта. Однако для обеспечения прямого доступа всем виртуальным машинам придется предоставить внутреннюю структуру данных. Поэтому в JNI для обращения к полям применяются специальные функции.

Поскольку в нашем примере поле `salary` имеет тип `double`, будем использовать функции `GetDoubleField()` и `SetDoubleField()`. Для полей данных других типов предусмотрены функции `GetIntField()` и `SetIntField()`, `GetObjectField()` и `SetObjectField()` и т.д. При работе с ними применяется следующий синтаксис:

```
x = (*env)->GetXxxField(env, this_obj, fieldID);
(*env)->SetXxxField(env, this_obj, fieldID, x);
```

Здесь параметр `fieldID` — значение специального типа, которое идентифицирует поле структуры, а суффикс *Xxx* — тип данных Java (например, `Object`, `Boolean`, `Byte` и т.д.). Для определения `fieldID` нужен объект `jclass`, получить который можно, используя функцию `GetObjectClass()` или `FindClass()`.

Функция `GetObjectClass` возвращает класс произвольного объекта:

```
jclass class_Employee = (*env)->GetObjectClass(env, this_obj);
```

Функция `FindClass` позволяет указать имя класса в виде строки (вместо точки для разделения пакетов нужно использовать символ /):

```
jclass class_String = (*env)->FindClass(env, "java/lang/String");
```

Для получения параметра `fieldID` следует применить функцию `GetFieldID()` с указанием имени поля и его *сигнатуры*, т.е. кодированного представления его типа. Например, для того чтобы определить идентификатор поля `salary`, нужно использовать выражение

```
jfieldID id_salary = (*env)->GetFieldID(env, class_Employee, "salary", "D");
```

Строка `"D"` обозначает тип `double`. Более подробно правила создания и использования сигнатур полей рассматриваются в следующем разделе.

На первый взгляд может показаться, что эти правила доступа к полям данных чрезмерно сложны. Но необходимо помнить о том, что создатели JNI для предоставления непосредственного доступа к полям должны были создать соответствующие `set`- и `get`-функции. Для сокращения накладных расходов при использовании этих функций вычисление идентификатора поля (что представляет собой наиболее сложную задачу) выделено в отдельную задачу. Таким образом, при многократном применении `set`- и `get`-функций для значения какого-то поля его идентификатор будет вычислен только один раз.

Итак, все описанные действия для метода `raiseSalary()` могут быть представлены приведенным ниже фрагментом кода.

```
JNIEXPORT void JNICALL Java_Employee_raiseSalary(JNIEnv* env,
  jobject this_obj, jdouble byPercent)
{
  /* Получение класса */
  jclass class_Employee = (*env)->GetObjectClass(env,
    this_obj);

  /* Получение идентификатора поля */
  jfieldID id_salary = (*env)->GetFieldID(env,
    class_Employee, "salary", "D");

  /* Получение значения поля */
  jdouble salary = (*env)->GetDoubleField(env, this_obj,
    id_salary);

  salary *= 1 + byPercent / 100;

  /* Установка значения поля */
  (*env)->SetDoubleField(env, this_obj, id_salary, salary);
}
```



Ссылки на классы действительны только до завершения платформенно-ориентированного метода. Поэтому значения, возвращаемые `GetObjectClass()`, не удастся кэшировать. Каждый раз при использовании платформенно-ориентированного метода необходимо вызывать `GetObjectClass()`. Если такое решение неприемлемо, вы можете заблокировать данную ссылку с помощью функции `NewGlobalRef()`, например, так, как показано ниже.

```
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 заблокированную ссылку на класс и идентификаторы полей можно использовать при последующих вызовах платформенно-ориентированного метода. После выполнения всех необходимых операций со ссылкой ее нужно разблокировать с помощью функции `DeleteGlobalRef()`.

```
(*env)->DeleteGlobalRef(env, class_X);
```

В листинге 12.11 приведен Java-код тестовой программы для класса `Employee`, в листинге 12.12 — код самого класса `Employee`, а в листинге 12.13 — код платформенно-ориентированного метода `raiseSalary()` на языке C.

Листинг 12.11. Содержимое файла `EmployeeTest.java`

```
1.  /**
2.   * @version 1.10 1999-11-13
3.   * @author Cay Horstmann
4.   */
```

```
5.  public class EmployeeTest
6.  {
7.    public static void main(String[] args)
8.    {
9.      Employee[] staff = new Employee[3];
10.
11.       staff[0] = new Employee("Harry Hacker", 35000);
12.       staff[1] = new Employee("Carl Cracker", 75000);
13.       staff[2] = new Employee("Tony Tester", 38000);
14.
15.       for (Employee e : staff)
16.       e.raiseSalary(5);
17.     for (Employee e : staff)
18.       e.print();
19.   }
20. }
```

Листинг 12.12. Содержимое файла `Employee.java`

```
1.  /**
2.   * @version 1.10 1999-11-13
3.   * @author Cay Horstmann
4.   */
5.
6.   public class Employee
7.   {
8.    public Employee(String n, double s)
9.    {
10.     name = n;
11.     salary = s;
12.   }
13.
14.   public native void raiseSalary(double byPercent);
15.
16.   public void print()
17.   {
18.     System.out.println(name + " " + salary);
19.   }
20.   private String name;
21.   private double salary;
22.
23.   static
24.   {
25.     System.loadLibrary("Employee");
26.   }
27. }
```

Листинг 12.13. Содержимое файла `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.  JNIEXPORT void JNICALL Java_Employee_raiseSalary(JNIEnv* env, jobject this_obj,
10.     jdouble byPercent)
11. {
12.    /* Получение класса */
13.    jclass class_Employee = (*env)->GetObjectClass(env, this_obj);
14.
15.    /* Получение идентификатора поля */
16.    jfieldID id_salary = (*env)->GetFieldID(env,
17.       class_Employee, "salary", "D");
18.
19.    /* Получение значения поля */
20.    jdouble salary = (*env)->GetDoubleField(env, this_obj,
21.       id_salary);
22.
23.    salary *= 1 + byPercent / 100;
24.
25.    /* Установка значения поля */
26.    (*env)->SetDoubleField(env, this_obj, id_salary, salary);
27. }
```

Получение доступа к статическим полям

Получение доступа к статическим полям осуществляется подобно получению доступа к нестатическим полям, а именно — с помощью функций `GetStaticFieldID()` и `GetStatic`*`Xxx`*`Field()`/`SetStatic`*`Xxx`*`Field()`, которые работают практически так же, как и их нестатические аналоги, но имеют два следующих отличия:

- из-за отсутствия объекта они требуют использовать для получения ссылки на класс не метод `GetObjectClass()`, а метод `FindClass()`;
- по той же причине, при получении доступа к полю они требуют указывать не объект экземпляра, а сам класс.

Например, ниже показано, как может быть получена ссылка на `System.out`:

```
/* Получение класса */
jclass class_System = (*env)->FindClass(env, "java/lang/System");

/* Получение идентификатора поля */
jfieldID id_out = (*env)->GetStaticFieldID(env, class_System,
  "out", "Ljava/io/PrintStream;");

/* Получение значения поля */
jobject obj_out = (*env)->GetStaticObjectField(env, class_System, id_out);
```

API Получение доступа к полям

- `jfieldID GetFieldID(JNIEnv *env, jclass cl, const char name[], const char fieldSignature[])`

  Возвращает идентификатор поля в классе.

- *`Xxx`* `Get`*`Xxx`*`Field(JNIEnv *env, jobject obj, jfieldID id)`

  Возвращает значение поля. Для типа поля *`Xxx`* может указываться одно из следующих значений: `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` или `Double`.

- `void Set`*`Xxx`*`Field(JNIEnv *env, jobject obj, jfieldID id,` *`Xxx`* `value)`
  Устанавливает для поля новое значение. Для типа поля *Xxx* может указываться одно из следующих значений: `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`* `GetStatic`*`Xxx`*`Field(JNIEnv *env, jclass cl, jfieldID id)`
  Возвращает значение статического поля. Для типа поля *Xxx* может указываться одно из следующих значений: `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` или `Double`.
- `void SetStatic`*`Xxx`*`Field(JNIEnv *env, jclass cl, jfieldID id,` *`Xxx`* `value)`
  Устанавливает для статического поля новое значение. Для типа поля *Xxx* может указываться одно из следующих значений: `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` или `Double`.

Кодирование сигнатуры

Для доступа к полям объекта и вызова методов, которые определены в языке Java, необходимо знать правила указания имен типов данных и сигнатуры методов. (Напомним, что сигнатура метода описывает параметры и возвращаемое значение метода.) Для этого используется приведенная ниже схема обозначений.

`B`	`byte`
`C`	`char`
`D`	`double`
`F`	`float`
`I`	`int`
`J`	`long`
`L`*`имя_класса`*`;`	тип класса
`S`	`short`
`V`	`void`
`Z`	`boolean`

Для описания типа массива применяется символ `[`. Например, массив строк описывается следующим образом:

```
[Ljava/lang/String;
```

Массив `float[][]`, соответственно, превращается в

```
[[F
```

Для полного представления сигнатуры метода сначала в круглых скобках перечисляются все типы параметров, а затем указывается возвращаемый тип. Например, сигнатура метода, получающего два целых числа и возвращающего одно целое число, будет выглядеть так:

```
(II)I
```

Метод `print()`, который мы использовали в предыдущем примере, будет иметь смешанную сигнатуру:

```
(Ljava/lang/String;)V
```

То есть он будет получать строку, а возвращать значение `void`.

Обратите внимание на то, что точка с запятой в конце выражения L*имя_класса* является признаком конца определяющего тип выражения, а не разделителем параметров. Например, возьмем следующий конструктор:

```
Employee(java.lang.String, double, java.util.Date)
```

Его сигнатура будет выглядеть так:

```
"(Ljava/lang/String;DLjava/util/Date;)V"
```

Обратите внимание на то, что между D и Ljava/util/Date; никакого разделителя нет. Кроме того, также еще обратите внимание и на то, что в этой системе обозначений для разделения имен пакетов и классов требуется использовать не символ точки (.), а символ косой черты (/). Символ V в конце обозначает, что возвращаемым типом является тип void. Хотя в Java возвращаемый тип для конструкторов и не указывается, символ V все равно должен добавляться к сигнатуре виртуальной машины.



Для автоматической генерации сигнатур методов из файлов классов можно применять команду javap с опцией -s, например, так:

```
javap -s -private Employee
```

Выполнение этой команды приведет к получению следующего вывода:

```
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
}
```



Никакого логического объяснения вынуждению программистов к применению именно такой смешанной схемы при описании сигнатур не существует. Разработчики механизма вызова платформенно-ориентированного кода точно так же легко могли бы написать функцию, умеющую считывать сигнатуры в стиле языка программирования Java, например, void(int,java.lang.String), и затем кодировать их в любое предпочитаемое ими внутреннее представление. С другой стороны, опять-таки, применение смешанных сигнатур позволяет познавать таинства программирования на уровне, более близком уровню виртуальной машины.

Вызов Java-методов

О том, что из Java можно вызывать функции, написанные на языке C, теперь вы уже знаете — в конце концов, ведь именно для этого и предназначены платформенно-ориентированные методы. А вот можно ли выполнять обратную операцию, то есть вызывать Java-методы из C-функций? И зачем вообще такая операция может быть нужна?

Ответ выглядит просто: довольно часто бывает так, что платформенно-ориентированному методу требуется запрашивать какую-то службу от переданного ему объекта. Поэтому в настоящем разделе мы покажем, как подобное можно делать, сначала с методами экземпляра, а потом — со статическими методами.

Методы экземпляра

Для получения примера вызова метода экземпляра на Java из платформенно-ориентированного кода усовершенствуем класс `Printf` и добавим метод, работающий подобно функции `fprintf()` на языке C. То есть этот метод должен быть способен выводить форматированную строку в произвольный поток `PrintWriter`. На языке Java определение этого метода выглядит так:

```
class Printf3
{
  public static native void fprint(PrintWriter out,
    String format, double x);
  ...
}
```

Сначала, как в реализованном ранее методе `sprint()`, составим строку, которая будет помещаться в объекте `str` типа `String`. Затем вызовем из функции C метод `print()` класса `PrintWriter`.

Любой Java-метод можно вызвать из C-функции с помощью приведенного ниже выражения:

```
(*env)->Call Xxx Method(env, неявный_параметр, идентификатор_метода, явные_параметры)
```

Здесь *Xxx* означает возвращаемый тип, например, `Void`, `Int`, `Obect` и т.д. Если при обращении к полю требовался идентификатор поля, то для вызова метода нужно использовать идентификатор метода. Для его получения предусмотрена JNI-функция `GetMethodID()` с параметрами, которые задают класс, имя метода и сигнатуру метода.

В нашем примере нужно получить идентификатор метода `print()` класса `PrintWriter`. Как показано в главе 12 первого тома, класс `PrintWriter` содержит девять разных методов, которые имеют одинаковое имя `print()`. Поэтому для выбора конкретного варианта нужно точно указать параметры и возвращаемое значение. Например, для `void print(java.lang.String)` сигнатура будет иметь вид `"(Ljava/lang/String;)V"`.

Ниже приведен фрагмент кода, который выполняет следующие действия.

1. Получает класс неявного параметра.
2. Получает идентификатор метода.
3. Вызывает метод.

```
/* Получение класса */
class_PrintWriter = (*env)->GetObjectClass(env, out);

/* Получение идентификатора метода */
id_print = (*env)->GetMethodID(env, class_PrintWriter,
  "print", "(Ljava/lang/String;)V");

/* Вызов метода */
(*env)->CallVoidMethod(env, out, id_print, str);
```

В листинге 12.14 приведен Java-код тестовой программы для класса `Printf3`, в листинге 12.15 — код самого класса `Printf3`, а в листинге 12.16 — код для платформенно-ориентированного метода `fprint()` на языке C.



Числовые идентификаторы методов и полей выполняют те же функции, что и объекты `Method` и `Field` в механизме отражения Java. Их можно взаимно преобразовывать с помощью следующих функций:

```
jobject ToReflectedMethod(JNIEnv* env, jclass class,
    jmethodID methodID); // Возвращает объект-метод.
methodID FromReflectedMethod(JNIEnv* env, jobject method);
jobject ToReflectedField(JNIEnv* env, jclass class,
    jfieldID fieldID);    // Возвращает объект-поле.
fieldID FromReflectedField(JNIEnv* env, jobject field);
```

Статические методы

Вызов статических методов из платформенно-ориентированного кода выполняется подобно вызову методов экземпляра, за исключением двух следующих отличий:

- для решения этой задачи используются методы `GetStaticMethodID()` и `CallStatic`*`Xxx`*`Method()`;
- при вызове метода вместо неявного параметра указывается класс.

В качестве примера рассмотрим следующий вызов статического метода `getProperty()`:

```
System.getProperty("java.class.path")
```

Возвращаемым значением этого метода является строка, содержащая текущий путь к классу.

Вначале необходимо найти класс. Из-за отсутствия каких-либо объектов `System` придется вместо метода `GetObjectClass()` использовать метод `FindClass()`:

```
jclass class_System = (*env)->FindClass(env, "java/lang/System");
```

Затем нужно получить идентификатор статического метода `getProperty()`, который имеет следующую сигнатуру:

```
"(Ljava/lang/String;)Ljava/lang/String;"
```

Таким образом, для получения идентификатора данного метода будет использоваться приведенный ниже оператор:

```
jmethodID id_getProperty = (*env)->GetStaticMethodID(env,
  class_System, "getProperty",
  "(Ljava/lang/String;)Ljava/lang/String;"));
```

И, наконец, необходимо выполнить сам вызов. Обратите внимание на то, что функции `CallStaticObjectMethod` передается объект, представляющий класс:

```
jobject obj_ret = (*env)->CallStaticObjectMethod(env,
  class_System, id_getProperty,
  (*env)->NewStringUTF(env, "java.class.path"));
```

Возвращаемое значение имеет тип `jobject`. Для выполнения строковых операций его потребуется привести к типу `jstring`:

```
jstring str_ret = (jstring)obj_ret;
```



В языке C типы `jstring` и `jclass`, а также типы массивов, которые рассматриваются ниже, эквивалентны `jobject`. Но в языке C++ для этих типов определена иерархия наследования. Например, в языке C++ допускается присваивание значения `jstring` переменной `jobject` без приведения типа, но приведение типа обязательно для присвоения значения `jobject` переменной `jstring`.

Конструкторы

Платформенно-ориентированный метод может создать новый объект Java за счет вызова его конструктора с помощью функции `NewObject()`:

```
jobject obj_new = (*env)->NewObject(env, класс,
  идентификатор_метода, параметры_конструктора);
```

Для получения идентификатора метода нужно использовать функцию `GetMethodID()` с указанием имени метода в виде строки `"<init>"` и закодированной сигнатуры конструктора с возвращаемым типом `void`. Например, покажем, как выглядит платформенно-ориентированный метод для создания объекта `FileOutputStream()`:

```
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: сигнатура конструктора описывает параметр типа `java.lang.String` и возвращаемое значение типа `void`.

Альтернативные вызовы методов

Существуют несколько вариантов JNI-функций, которые позволяют вызывать Java-методы из платформенно-ориентированного кода. Они являются не таким важными, как те, которые уже были рассмотрены ранее, но иногда могут быть очень полезны.

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, минуя обычный динамический механизм управления вызовами.

Все такие функции имеют варианты с суффиксами `A` или `V`, которые получают явно заданные параметры в массиве или структуре типа `va_list` (она определена в файле заголовков `stdarg.h` языка C).

Листинг 12.14. Содержимое файла `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. Содержимое файла `Printf3.java`

```
1.  import java.io.*;
2.
3.  /**
4.   * @version 1.10 1997-07-01
5.   * @author Cay Horstmann
6.   */
7.  class Printf3
8.  {
9.    public static native void fprint(PrintWriter out,
10.     String format, double x);
11.
12.   static
13.   {
14.     System.loadLibrary("Printf3");
15.   }
16. }
```

Листинг 12.16. Содержимое файла `Printf3.c`

```
1.  /**
2.   * @version 1.10 1997-07-01
3.   * @author Cay Horstmann
4.   */
5.
6.  #include "Printf3.h"
7.  #include <string.h>
8.  #include <stdlib.h>
9.  #include <float.h>
10. /**
11.  * @param format Строка, содержащая спецификатор формата
12.  * (например, "%8.2f"). Подстроки "%%" пропускаются.
13.  * @return Указатель на спецификатор формата или NULL
14.  */
15. char* find_format(const char format[])
16. {
17.   char* p;
18.   char* q;
19.   p = strchr(format, '%');
20.   while (p != NULL && *(p + 1) == '%') /* Пропуск %% */
21.     p = strchr(p + 2, '%');
22.   if (p == NULL) return NULL;
23.   /* Проверка уникальности % */
24.   p++;
25.   q = strchr(p, '%');
26.   while (q != NULL && *(q + 1) == '%') /* Пропуск %% */
27.     q = strchr(q + 2, '%');
28.   if (q != NULL) return NULL; /* % не уникальный */
29.   q = p + strspn(p, " -0+#"); /* Пропуск флагов */
30.   q += strspn(q, "0123456789"); /* Пропуск ширины поля */
31.   if (*q == '.') { q++; q += strspn(q, "0123456789"); }
32.     /* Пропуск точности */
33.   if (strchr("eEfFgG", *q) == NULL) return NULL;
34.     /* Формат, отличный от формата с плавающей точкой */
35.   return p;
36. }
```

```
37.  JNIEXPORT void JNICALL Java_Printf3_fprint(JNIEnv* env, jclass cl,
38.    jobject out, jstring format, jdouble x)
39.  {
40.    const char* cformat;
41.    char* fmt;
42.    jstring str;
43.    jclass class_PrintWriter;
44.    jmethodID id_print;
45.
46.    cformat = (*env)->GetStringUTFChars(env, format, NULL);
47.    fmt = find_format(cformat);
48.    if (fmt == NULL)
49.      str = format;
50.    else
51.    {
52.      char* cstr;
53.      int width = atoi(fmt);
54.      if (width == 0) width = DBL_DIG + 10;
55.      cstr = (char*) malloc(strlen(cformat) + width);
56.      sprintf(cstr, cformat, x);
57.      str = (*env)->NewStringUTF(env, cstr);
58.      free(cstr);
59.    }
60.    (*env)->ReleaseStringUTFChars(env, format, cformat);
61.
62.    /* Обращение к ps.print(str) */
63.
64.    /* Получение класса */
65.    class_PrintWriter = (*env)->GetObjectClass(env, out);
66.
67.    /* Получение идентификатора метода */
68.    id_print = (*env)->GetMethodID(env, class_PrintWriter,
69.      "print", "(Ljava/lang/String;)V");
70.
71.    /* Вызов метода */
72.    (*env)->CallVoidMethod(env, out, id_print, str);
73.  }
```

API Вызов Java-методов

- `jmethodID GetMethodID(JNIEnv *env, jclass cl, const char name[], const char methodSignature[])`

  Возвращает идентификатор метода в классе.

- *Xxx* `Call`*Xxx*`Method(JNIEnv *env, jobject obj, jmethodID id, args)`
- *Xxx* `Call`*Xxx*`MethodA(JNIEnv *env, jobject obj, jmethodID id, jvalue args[])`
- *Xxx* `Call`*Xxx*`MethodV(JNIEnv *env, jobject obj, jmethodID id, va_list args)`

  Вызывает метод. Для возвращаемого типа *Xxx* может указываться одно из следующих значений: `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` или `Double`. Первая функция принимает переменное количество параметров; они просто присоединяются после идентификатора метода. Вторая функция получает параметры метода в виде массива значений `jvalue`, который имеет следующий вид:

```
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`, определение которой содержится в заголовочном файле `stdarg.h`.

- *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)`

Вызывает метод, минуя механизм динамического распределения. Для возвращаемого типа *Xxx* может указываться одно из следующих значений: `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` или `Double`. Первая функция имеет переменное количество аргументов; они просто присоединяются после идентификатора метода. Второй функции параметры метода передаются в массиве `jvalue`. Третья функция получает параметры в виде структуры `va_list` в соответствии с определением, содержащимся заголовочном C-файле `stdarg.h`.

- `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)`

Вызывает статический метод. Для возвращаемого типа может указываться одно из следующих значений: `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` или `Double`. Первая функция имеет переменное количество параметров, которые просто присоединяются после идентификатора метода. Вторая функция принимает параметры метода в виде массива `jvalue`, а третья — в виде структуры `va_list` в соответствии с определением, содержащимся в заголовочном C-файле `stdarg.h`.

- `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)`

Вызывает конструктор класса. Идентификатор метода получается из `GetMethodID()` с именем метода в виде строки `"<init>"` и возвращаемым типом `void`. Первая функция принимает переменное количество параметров метода, которые просто присоединяются после идентификатора метода. Вторая функция получает параметры метода в виде массив `jvalue`, а третья — в виде структуры `va_list` в соответствии с определением, содержащимся в заголовочном C-файле `stdarg.h`.

Доступ к элементам массивов

Все типы массивов в языке Java имеют соответствующие им типы массивов в языке C, которые перечислены в табл. 12.2.

Таблица 12.2. Соответствие между типами массивов в языках Java и C

Java	C
`boolean[]`	`jbooleanArray`
`byte[]`	`jbyteArray`
`char[]`	`jcharArray`
`int[]`	`jintArray`
`short[]`	`jshortArray`
`long[]`	`jlongArray`
`float[]`	`jfloatArray`
`double[]`	`jdoubleArray`
`Object[]`	`jobjectArray`



В языке C все типы массивов фактически являются синонимами типа `jobject`. В языке C++, однако, они упорядочены в иерархическую структуру, как показано на рис. 12.3. Тип `jarray` обозначает универсальный массив.

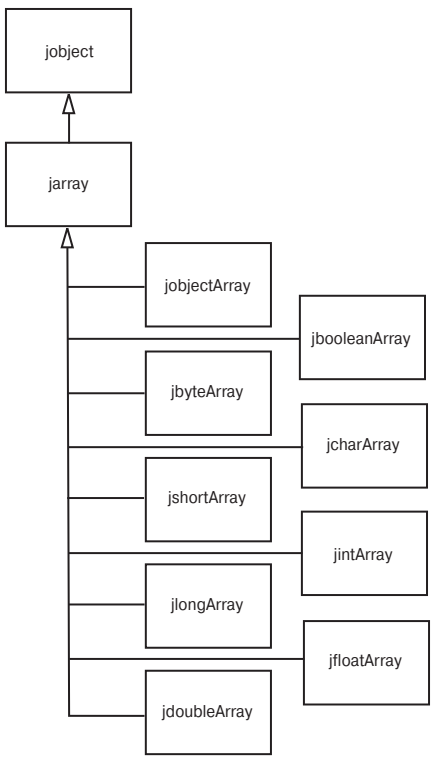


Рис. 12.3. Иерархия наследования типов массивов в языке C++

Функция `GetArrayLength()` возвращает длину массива:

```
jarray array = ...;
jsize length = (*env)->GetArrayLength(env, array);
```

Способ доступа к элементам массива зависит от того, хранятся в нем объекты или значения простых типов (числа, символы или логические значения). Для доступа к элементам массива объектов используются методы `GetObjectArrayElement()` и `SetObjectArrayElement()`:

```
jobjectArray array = ...;
int i, j;
jobject x = (*env)->GetObjectArrayElement(env, array, i);
(*env)->SetObjectArrayElement(env, array, j, x);
```

Несмотря на простоту, этот подход крайне неэффективен. Гораздо лучше иметь непосредственный доступ к данным; особенно это важно при выполнении операций с векторами или матрицами.

Функция `Get`*`Xxx`*`ArrayElements()` возвращает C-указатель на начальный элемент массива. Как и при использовании обычных строк, по окончании работы с этим указателем нужно вызывать функцию `Release`*`Xxx`*`ArrayElements()`. Здесь *Xxx* обозначает простой тип, а не объект. С помощью C-указателя можно непосредственно считывать и записывать значения элементов массива. Однако указатель *может указывать на копию массива*, поэтому любые изменения будут отражены в исходном массиве только после вызова функции `Release`*`Xxx`*`ArrayElements()`.



Чтобы определить, работаете вы с исходным массивом или с его копией, нужно указать в качестве третьего параметра функции `Get`*`Xxx`*`ArrayElements()` переменную типа `jboolean`. Если используется копия массива, то переменная будет содержать значение `JNI_TRUE`. Если эта информация вас не интересует, то в качестве третьего параметра функции `Get`*`Xxx`*`ArrayElements()` следует передать значение `NULL`.

Ниже приведен фрагмент кода, в котором все элементы массива типа `double` умножаются на постоянное значение `scaleFactor` типа `double`. Доступ к отдельным элементам `a[i]` осуществляется с помощью C-указателя:

```
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 "сборщики мусора" перемещают объекты и обновляют ссылки на объекты. Эта стратегия не совместима с подходом, предполагающим фиксацию массива по определенному адресу, поскольку сборщик мусора не может обновлять значения указателей в платформенно-ориентированном коде.



В виртуальной машине Sun JVM массив значений типа `boolean` представляется в виде массива, содержащего 32-битные значения. Метод `GetBooleanArrayElements()` распаковывает его и копирует в массив `jboolean`.

Если необходим доступ лишь к нескольким элементам большого массива, следует использовать методы `Get`*`Xxx`*`ArrayRegion()` и `Set`*`Xxx`*`ArrayRegion()`. Они копируют элементы Java-массива, находящиеся в заданном диапазоне индексов, в C-массив и обратно.

Новые Java-массивы в платформенно-ориентированных методах создаются с помощью функции `New`*`Xxx`*`Array()`. Для создания нового массива объектов нужно указать его длину, тип элементов массива, а также исходное значение для всех элементов (как правило, `NULL`):

```
jclass class_Employee = (*env)->FindClass(env, "Employee");
jobjectArray array_e = (*env)->NewObjectArray(env, 100,
  class_Employee, NULL);
```

Массивы элементов простых типов устроены проще; для них достаточно указать длину массива:

```
jobjectArray array_d = (*env)->NewDoubleArray(env, 100);
```

Такой массив заполняется нулями.



В Java SE 1.4 в API JNI появились три дополнительных метода:

```
jobject NewDirectByteBuffer(JNIEnv *env, void* address, jlong capacity)
void* GetDirectBufferAddress(JNIEnv *env, jobject buf)
jlong GetDirectBufferCapacity(JNIEnv *env, jobject buf)
```

Непосредственные буферы (direct buffer) применяются в пакете `java.nio` для поддержки более эффективных операций ввода-вывода и сведения к минимуму числа операций по копированию данных между платформенно-ориентированными массивами и массивами 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 для элемента массива новое значение.

- *`Xxx`*`* Get`*`Xxx`*`ArrayElements(JNIEnv *env, jarray array, jboolean* isCopy)`

  Выдает C-указатель на элементы массива Java. Для типа поля *`Xxx`* может указываться одно из следующих значений: `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` или `Double`. После окончания работы с полученным указателем он должен передаваться функции `Release`*`Xxx`*`ArrayElements()`. Параметр `isCopy` может либо содержать значение `NULL`, либо указывать на переменную типа `jboolean`, которая в случае создания копии заполняется значением `JNI_TRUE`, а в противном случае — значением `JNI_TRUE`.

- `void Release`*`Xxx`*`ArrayElements(JNIEnv *env, jarray array, `*`Xxx`*` elems[], jint mode)`

  Информирует виртуальную машину Java о том, что указатель, полученный с помощью функции `Get`*`Xxx`*`ArrayElements()`, уже больше не требуется. Параметр

mode может иметь одно из следующих значений: 0 (указывающее освободить буфер elems после обновления элементов массива), JNI_COMMIT (указывающее не освобождать буфер elems после обновления элементов массива) или JNI_ABORT (указывающее освобождать буфер elems без обновления элементов массива).

- void Get*Xxx*ArrayRegion(JNIEnv *env, jarray array, jint start, jint length, *Xxx* elems[])

  Копирует элементы из массива на языке Java в массив на языке C. Для типа поля *Xxx* может указываться одно из следующих значений: Boolean, Byte, Char, Short, Int, Long, Float или Double.

- void Set*Xxx*ArrayRegion(JNIEnv *env, jarray array, jint start, jint length, *Xxx* elems[])

  Копирует элементы из массива на языке C в массив на языке Java. Для типа поля *Xxx* может указываться одно из следующих значений: Boolean, Byte, Char, Short, Int, Long, Float или Double.

Обработка ошибок

Использование платформенно-ориентированных методов представляет существенную угрозу безопасности Java-программ. 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 платформенно-ориентированные методы, необходимо уделить особое внимание обработке подобных ошибок. В частности, если платформенно-ориентированный метод обнаруживает какую-то проблему, с которой не может справиться, 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 при завершении платформенно-ориентированного метода.

Если вы хотите использовать функцию Throw, следует вызвать функцию NewObject() для создания экземпляра класса, являющегося подклассом Throwable. Например, в приведенном ниже фрагменте кода создается объект 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 по умолчанию */
jthrowable obj_exc = (*env)->NewObject(env, class_EOFException,
  id_EOFException);
(*env)->Throw(env, obj_exc);
```

Однако на практике удобнее использовать функцию ThrowNew(), которая также создает объект исключения. Исходной информацией для нее является байтовая последовательность в модифицированном формате UTF-8:

```
(*env)->ThrowNew(env,
  (*env)->FindClass(env, "java/io/EOFException"),
  "Unexpected end of file");
```

Обе функции Throw и ThrowNew *всего лишь* передают исключение, а не прерывают поток управления платформенно-ориентированного метода. Поэтому виртуальная машина Java может генерировать исключение только после окончания работы мето-

да. Это значит, что после каждого вызова функции `Throw()` или `ThrowNew()` должен быть указан оператор `return`.



При реализации платформенно-ориентированых методов на языке C++ в теле метода нельзя генерировать Java-исключение. Теоретически можно было бы выполнить преобразование исключений между языками C++ и Java, однако в настоящее время эта возможность не реализована. Для генерации Java-объектов исключений в платформенно-ориентированном коде на языке C++ придется использовать функции `Throw()` и `ThrowNew()`; кроме того, следует позаботиться, чтобы платформенно-ориентированные методы не генерировали исключения C++.

Обычно платформенно-ориентированный код не должен заниматься обработкой Java-исключений, однако подобная ситуация может возникнуть при вызове из платформенно-ориентированной функции Java-метода.

Например, функция `SetObjectArrayElement()` генерирует исключение `ArrayIndexOutOfBoundsException`, если значение индекса выходит за границы массива, а также исключение `ArrayStoreException`, если класс хранимого объекта не является подклассом для элемента массива. В таких ситуациях платформенно-ориентированный метод должен вызывать функцию `ExceptionOccured()`, чтобы определить наличие исключения. В результате приведенного ниже вызова возвращается ссылка на текущий объект исключения. Если же ни одного исключения в очереди нет, функция возвращает значение `NULL`.

```
jtrowable obj_exc = (*env)->ExceptionOccured(env);
```

Для проверки наличия генерированного исключения без получения ссылки на него можно использовать функцию `ExceptionCheck()`:

```
jboolean occured = (*env)->ExceptionCheck(env);
```

Обычно при генерации исключения платформенно-ориентированный метод лишь возвращает управление. В результате виртуальная машина Java может делегировать исключение соответствующему фрагменту Java-программы. Однако платформенно-ориентированный метод *может* проанализировать объект исключения и определить, сумеет ли он обработать его своими силами. Если такая обработка возможна, необходимо отозвать сообщение об исключении с помощью следующей функции:

```
(*env)->ExceptionClear(env);
```

В примере программы, код которой приведен в конце раздела, реализуется платформенно-ориентированный метод `fprintf()` с некоторыми чрезмерными на первый взгляд, но весьма полезными на практике мерами по обработке следующих исключений:

- `NullPointerException`, если указатель строки форматирования содержит значение `NULL`;
- `IllegalArgumentException`, если строка форматирования не содержит спецификатор `%`, необходимый для вывода значений типа `double`;
- `OutOfMemoryError`, если произошла ошибка при выполнении функции `malloc()`.

И, наконец, для демонстрации того, как может выполняться проверка на предмет наличия исключения при вызове Java-метода из платформенно-ориентированного кода, осуществляется отправка строки в поток, по одному символу за раз, с вызовом `ExceptionOccured()` после вывода каждого символа. В листинге 12.17 приведен код

платформенно-ориентированного метода, а в листинге 12.18 — определение класса, использующего этот метод. Обратите внимание на то, что в случае возникновения исключения при вызове функции `PrintWriter.print()` платформенно-ориентированный метод прекращает свою работу не сразу: сначала он освобождает буфер `cstr`. Когда же платформенно-ориентированный метод возвращается, виртуальная машина Java снова генерирует исключение. Тестовая программа, приведенная в листинге 11.19, позволяет увидеть, как именно платформенно-ориентированный метод выдает исключение в случае, если строка форматирования оказывается недействительной.

Листинг 12.17. Содержимое файла `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 Строка, содержащая спецификатор формата
13.  * (например, "%8.2f"). Подстроки "%%" пропускаются.
14.  * @return Указатель на спецификатор формата (без учета '%')
15.  * или NULL, если уникального спецификатора формата не было.
16.  */
17. char* find_format(const char format[])
18. {
19.    char* p;
20.    char* q;
21.
22.    p = strchr(format, '%');
23.    while (p != NULL && *(p + 1) == '%')   /* Пропуск %% */
24.       p = strchr(p + 2, '%');
25.    if (p == NULL) return NULL;
26.    /* Проверка единственности %  */
27.    p++;
28.    q = strchr(p, '%');
29.    while (q != NULL && *(q + 1) == '%')   /* Пропуск %% */
30.       q = strchr(q + 2, '%');
31.    if (q != NULL) return NULL;            /* % не единственный */
32.    q = p + strspn(p, " -0+#");            /* Пропуск флагов */
33.    q += strspn(q, "0123456789");          /* Пропуск ширины поля */
34.    if (*q == '.') { q++; q += strspn(q, "0123456789"); }
35.       /* Пропуск точности */
36.    if (strchr("eEfFgG", *q) == NULL) return NULL;
37.       /* Формат, отличный от формата с плавающей точкой */
38.    return p;
39. }
40.
41. JNIEXPORT void JNICALL Java_Printf4_fprint(JNIEnv* env, jclass cl,
42.    jobject out, jstring format, jdouble x)
43. {
44.    const char* cformat;
45.    char* fmt;
46.    jclass class_PrintWriter;
```

```
47.    jmethodID id_print;
48.    char* cstr;
49.    int width;
50.    int i;
51.    if (format == NULL)
52.    {
53.       (*env)->ThrowNew(env,
54.          (*env)->FindClass(env,
55.          "java/lang/NullPointerException"),
56.          "Printf4.fprint: format is null");
57.       return;
58.    }
59.
60.    cformat = (*env)->GetStringUTFChars(env, format, NULL);
61.    fmt = find_format(cformat);
62.
63.    if (fmt == NULL)
64.    {
65.       (*env)->ThrowNew(env,
66.          (*env)->FindClass(env,
67.          "java/lang/IllegalArgumentException"),
68.          "Printf4.fprint: format is invalid");
69.       return;
70.    }
71.
72.    width = atoi(fmt);
73.    if (width == 0) width = DBL_DIG + 10;
74.    cstr = (char*)malloc(strlen(cformat) + width);
75.
76.    if (cstr == NULL)
77.    {
78.       (*env)->ThrowNew(env,
79.          (*env)->FindClass(env, "java/lang/OutOfMemoryError"),
80.          "Printf4.fprint: malloc failed");
81.       return;
82.    }
83.
84.    sprintf(cstr, cformat, x);
85.
86.    (*env)->ReleaseStringUTFChars(env, format, cformat);
87.
88.    /* Обращение к ps.print(str) */
89.
90.    /* Получение класса */
91.    class_PrintWriter = (*env)->GetObjectClass(env, out);
92.
93.    /* Получение идентификатора метода */
94.    id_print = (*env)->GetMethodID(env, class_PrintWriter,
95.       "print", "(C)V");
96.
97.    /* Вызов метода */
98.    for (i = 0; cstr[i] != 0 && !(*env)->ExceptionOccurred(env);
99.        i++)
100.      (*env)->CallVoidMethod(env, out, id_print, cstr[i]);
101.
102.   free(cstr);
103. }
```

Листинг 12.18. Содержимое файла `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,
10.     String format, double x);
11.
12.   static
13.   {
14.     System.loadLibrary("Printf4");
15.   }
16. }
```

Листинг 12.19. Содержимое файла `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.     /*  Этот вызов генерирует исключение
16.       из-за неверного формата - %% */
17.     Printf4.fprint(out, "Amount due = %%8.2f\n", amountDue);
18.     out.flush();
19.   }
20. }
```

API Обработка исключений Java

- `jint Throw(JNIEnv *env, jthrowable obj)`

  Подготавливает исключение, которое должно выдаваться после выхода из платформенно-ориентированного кода. В случае успешного выполнения этого кода возвращает `0`, а в случае неудачного выполнения — отрицательное значение.

- `jint ThrowNew(JNIEnv *env, jclass cl, const char msg[])`

  Подготавливает исключение типа `cl`, которое должно выдаваться при выходе из платформенно-ориентированного кода. В случае успешного выполнения возвращает `0`, а в случае неудачного выполнения — отрицательное значение. `msg` — строка в модифицированном формате UTF-8, обозначающая параметр конструктора `String` объекта исключения.

- `jthrowable ExceptionOccurred(JNIEnv *env)`

  Возвращает объект исключения, если исключение еще находится на этапе подготовки, или значение `NULL`, если такого объекта нет.

- `jboolean ExceptionCheck(JNIEnv *env)`

  Возвращает значение `true`, если исключение все еще находится на этапе подготовки.

- `void ExceptionClear(JNIEnv *env)`

  Удаляет из очереди подготовки все объекты исключений.

API вызова

До сих пор рассматривались только программы на языке Java, делавшие несколько вызовов на языке C, предположительно из-за того, что либо код на языке C мог выполняться быстрее, либо из-за того, что это открывало доступ к таким возможностям, которые были не доступны из Java. Теперь давайте предположим, что возникла обратная ситуация, то есть, что имеется программа на языке C или C++ и требуется организовать из нее несколько вызовов Java-кода. Включать виртуальную машину Java в программу на языке C или C++ позволяет специально предназначенный для этого *API-интерфейс вызова* (invocation API). Минимальный фрагмент кода, который необходимо использовать для инициализации виртуальной машины Java, выглядит следующим образом:

```
JavaVMOption options[2];
JavaVMInitArgs vm_args;
JavaVM *jvm;
JNIEnv *env;
long status;
...

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;

status = JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);
```

Вызов `JNI_CreateJavaVM()` приводит к созданию виртуальной машины и добавлению указателя на нее — `jvm` и на среду выполнения — `env`.

Количество опций для виртуальной машины Java можно указывать любое, просто увеличивая размер массива `options` и значение параметра `vm_args.nOptions`. Например, использование показанной ниже команды:

```
options[i].optionString = "-Djava.compiler=NONE";
```

приведет к отключению оперативного компилятора.



В случае возникновения проблем, например, аварийного завершения работы программы, отказа программы инициализировать виртуальную машину Java или невозможности загрузить классы, можно включать режим отладки. Это делается следующим образом:

```
options[i].optionString = "-verbose:jni";
```

В этом режиме отображаются сообщения с информацией о ходе процесса инициализации JVM. В случае отсутствия сообщений о выполнении загрузки классов следует проверять как общий параметр пути, так и параметры пути к классам.

После настройки виртуальной машины Java можно начинать вызывать Java-методы описанным в предыдущих разделах образом, то есть просто путем использования указателя env.

Указатель jvm следует использовать только при необходимости вызова из API-интерфейса вызова других функций. В настоящее время существует всего лишь четыре таких функции. Наиболее важной из них является функция DestroyJavaVM(), предназначенная для прекращения работы виртуальной машины Java:

```
(*jvm)->DestroyJavaVM(jvm);
```

К сожалению, в Windows стало сложно осуществлять динамическое связывание с функцией JNI_CreateJavaVM из библиотеки jre/bin/client/jvm.dll из-за изменения правил связывания в Vista и зависимости Sun от более старой библиотеки исполняющей среды C. В нашей демонстрационной программе данная проблема решается путем выполнения загрузки упомянутой библиотеки вручную. Точно такой же подход применяется и в программе java; убедиться в этом можно, взглянув на launcher/java_md.c в файле src.jar, который поставляется в виде части JDK.

Приведенная в листинге 12.20 демонстрационная программа на языке C сначала инициализирует виртуальную машину Java, а затем вызывает метод main() класса Welcome, который рассматривался в главе 2 первого тома книги. (Перед запуском этой программы нужно обязательно скомпилировать файл Welcome.java.)

Листинг 12.20. Содержимое файла 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
```

```
33.    options[0].optionString = "-Djava.class.path=.";
34.
35.    memset(&vm_args, 0, sizeof(vm_args));
36.    vm_args.version = JNI_VERSION_1_2;
37.    vm_args.nOptions = 1;
38.    vm_args.options = options;
39.
40.  #ifdef _WINDOWS
41.    hjvmlib = loadJVMLibrary();
42.    createJavaVM = (CreateJavaVM_t) GetProcAddress(hjvmlib, "JNI_CreateJavaVM");
43.    status = (*createJavaVM)(&jvm, (void **) &env, &vm_args);
44.  #else
45.    status = JNI_CreateJavaVM(&jvm, (void **) &env, &vm_args);
46.  #endif
47.
48.    if (status == JNI_ERR)
49.    {
50.       fprintf(stderr, "Error creating VM\n");
51.                    // Ошибка при создании VM
52.       return 1;
53.    }
54.
55.    class_Welcome = (*env)->FindClass(env, "Welcome");
56.    id_main = (*env)->GetStaticMethodID(env, class_Welcome,
57.       "main", "([Ljava/lang/String;)V");
58.
59.    class_String = (*env)->FindClass(env, "java/lang/String");
60.    args = (*env)->NewObjectArray(env, 0, class_String, NULL);
61.    (*env)->CallStaticVoidMethod(env, class_Welcome,
62.       id_main, args);
63.
64.    (*jvm)->DestroyJavaVM(jvm);
65.
66.    return 0;
67.  }
68.
69.  #ifdef _WINDOWS
70.
71.  static int GetStringFromRegistry(HKEY key, const char *name, char *buf, jint bufsize)
72.  {
73.    DWORD type, size;
74.
75.    return RegQueryValueEx(key, name, 0, &type, 0, &size) == 0
76.      && type == REG_SZ
77.      && size < (unsigned int) bufsize
78.      && RegQueryValueEx(key, name, 0, 0, buf, &size) == 0;
79.  }
80.
81.  static void GetPublicJREHome(char *buf, jint bufsize)
82.  {
83.    HKEY key, subkey;
84.    char version[MAX_PATH];
85.
86.    /* Поиск текущей версии JRE. */
87.    char *JRE_KEY = "Software\\JavaSoft\\Java Runtime Environment";
88.    if (RegOpenKeyEx(HKEY_LOCAL_MACHINE, JRE_KEY, 0, KEY_READ, &key) != 0)
89.    {
```

```
    fprintf(stderr, "Error opening registry key '%s'\n", JRE_KEY);
                // Ошибка при открытии ключа реестра
    exit(1);
  }

  if (!GetStringFromRegistry(key, "CurrentVersion", version, sizeof(version)))
  {
    fprintf(stderr, "Failed reading value of registry
        key:\n\t%s\\CurrentVersion\n", JRE_KEY);
                // Не удалось прочитать значение реестра
    RegCloseKey(key);
    exit(1);
  }

  /* Поиск каталога, в котором установлена текущая версия. */
  if (RegOpenKeyEx(key, version, 0, KEY_READ, &subkey) != 0)
  {
    fprintf(stderr, "Error opening registry key '%s\\%s'\n", JRE_KEY, version);
                // Ошибка при открытии ключа реестра
      RegCloseKey(key);
      exit(1);
  }

  if (!GetStringFromRegistry(subkey, "JavaHome", buf, bufsize))
  {
    fprintf(stderr, "Failed reading value of registry
        key:\n\t%s\\%s\\JavaHome\n",
                 // Не удалось прочитать значение реестра
      JRE_KEY, version);
    RegCloseKey(key);
    RegCloseKey(subkey);
    exit(1);
  }

  RegCloseKey(key);
  RegCloseKey(subkey);
}

static HINSTANCE loadJVMLibrary(void)
{
  HINSTANCE h1, h2;
  char msvcdll[MAX_PATH];
  char javadll[MAX_PATH];
  GetPublicJREHome(msvcdll, MAX_PATH);
  strcpy(javadll, msvcdll);
  strncat(msvcdll, "\\bin\\msvcr71.dll", MAX_PATH - strlen(msvcdll));
  msvcdll[MAX_PATH - 1] = '\0';
  strncat(javadll, "\\bin\\client\\jvm.dll", MAX_PATH - strlen(javadll));
  javadll[MAX_PATH - 1] = '\0';

  h1 = LoadLibrary(msvcdll);
  if (h1 == NULL)
  {
    fprintf(stderr, "Can't load library msvcr71.dll\n");
                // Не удается загрузить библиотеку msvcr71.dll
    exit(1);
  }
```

```
147.   h2 = LoadLibrary(javadll);
148.   if (h2 == NULL)
149.   {
150.     fprintf(stderr, "Can't load library jvm.dll\n");
151.                 // Не удается загрузить библиотеку jvm.dll
152.     exit(1);
153.   }
154.   return h2;
155. }
156.
157. #endif
```

Чтобы скомпилировать эту программу в операционной системе Linux, нужно использовать такую команду:

```
gcc -I jdk/include -I jdk/include/linux -o InvocationTest
    -L jdk/jre/lib/i386/client -ljvm InvocationTest.c
```

В операционной системе Solaris, применяется следующая команда:

```
cc -I jdk/include -I jdk/include/solaris -o InvocationTest
   -L jdk/jre/lib/sparc -ljvm InvocationTest.c
```

При желании же скомпилировать эту программу в операционной системе Windows с помощью компилятора Microsoft, нужно использовать такую командную строку:

```
cl -D_WINDOWS -I jdk\include -I jdk\include\win32 InvocationTest.c
   jdk\lib\jvm.lib advapi32.lib
```

Придется обязательно проверять, чтобы переменные окружения `INCLUDE` и `LIB` включали путь к заголовочным файлам и файлам библиотек API-интерфейса Windows.

В среде Cygwin эту программу можно скомпилировать с помощью показанной ниже команды:

```
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 этой программы в Linux/UNIX следует обязательно удостовериться, что в `LD_LIBRARY_PATH` содержатся каталоги для совместно используемых библиотек. Например, в случае использования оболочки `bash` и операционной системы Linux для этого потребуется выполнить следующую команду:

```
export LD_LIBRARY_PATH=jdk/jre/lib/i386/client:$LD_LIBRARY_PATH
```

API Функции API запуска

- `jint JNI_CreateJavaVM(JavaVM** p_jvm, void** p_env, JavaVMInitArgs* vm_args)`

  Инициализирует виртуальную машину Java. В случае успешного завершения эта функция возвращает значение `0`, в противном случае — значение `JNI_ERR`.

Параметры:	`p_jvm`	Указатель на таблицу функций запуска
	`p_env`	Указатель на таблицу функций JNI
	`vm_args`	Параметры виртуальной машины

- `jint DestroyJavaVM(JavaVM* jvm)`

  Удаляет виртуальную машину Java. При успешном завершении эта функция возвращает значение `0`, в противном случае — отрицательное значение. Данную функцию можно вызывать с помощью указателя виртуальной машины, например, `(*jvm)->DestroyJavaVM(jvm)`.

Пример обращения к реестру Windows

В данном разделе рассматривается пример использования всех перечисленных в этой главе способов работы с платформенно-ориентированными методами для обработки строк, массивов, объектов, вызова конструкторов и обработки исключений. Основная цель — описать создание Java-оболочки для подмножества функций API, реализованных на языке C и предназначенных для работы с реестром операционной системы Windows. Из-за применения в ней функций для работы с объектом, специфическим для конкретной среды, т.е. реестром Windows, она, конечно, не может быть перенесена на другие платформы. Именно поэтому в языке Java не предусмотрено никаких инструментов для работы с реестром, а предполагается, что разработчики будут использовать для этого платформенно-ориентированный код.

Общие сведения о реестре Windows

Как известно, реестр представляет собой хранилище конфигурационных параметров операционной системы Windows и прикладных программ. Наличие такого хранилища очень удобно для администрирования, создания резервных копий и настройки приложений. Недостатком реестра является то, что он сам по себе становится критичным элементом. При наличии ошибки в реестре вся система будет работать с ошибками или даже не загрузится.

Использовать реестр для хранения конфигурационных параметров приложений на языке Java не рекомендуется. Для этого лучше использовать API для поддержки свойств (подробнее данный вопрос рассматривается в главе 10 первого тома). В данном случае реестр используется лишь для демонстрации включения платформенно-ориентированных функций в Java-класс.

Основным инструментом для работы с реестром является *редактор реестра* (registry editor). Его применение чревато большими неприятностями для начинающих пользователей, поэтому для его запуска не предусмотрено никаких визуальных элементов пользовательского интерфейса. Чтобы вызывать редактор реестра в режиме командной строки (или в диалоговом окне, которое появится на экране после выбора пункта меню Start⇨Run (Пуск⇨Выполнить)), нужно ввести команду `regedit`. На рис. 12.4 показан внешний вид окна редактора реестра.

В левой части окна редактора реестра представлены ключи, упорядоченные в виде иерархической древовидной структуры. Обратите внимание, что каждый ключ находится в поддереве, начинающемся в одном из перечисленных ниже узлов с префиксом `HKEY`:

```
HKEY_CLASSES_ROOT
HKEY_CURENT_USER
HKEY_LOCAL_MACHINE
...
```

В правой части окна редактора реестра отображаются пары “имя=значение”, ассоциируемые с конкретным ключом.

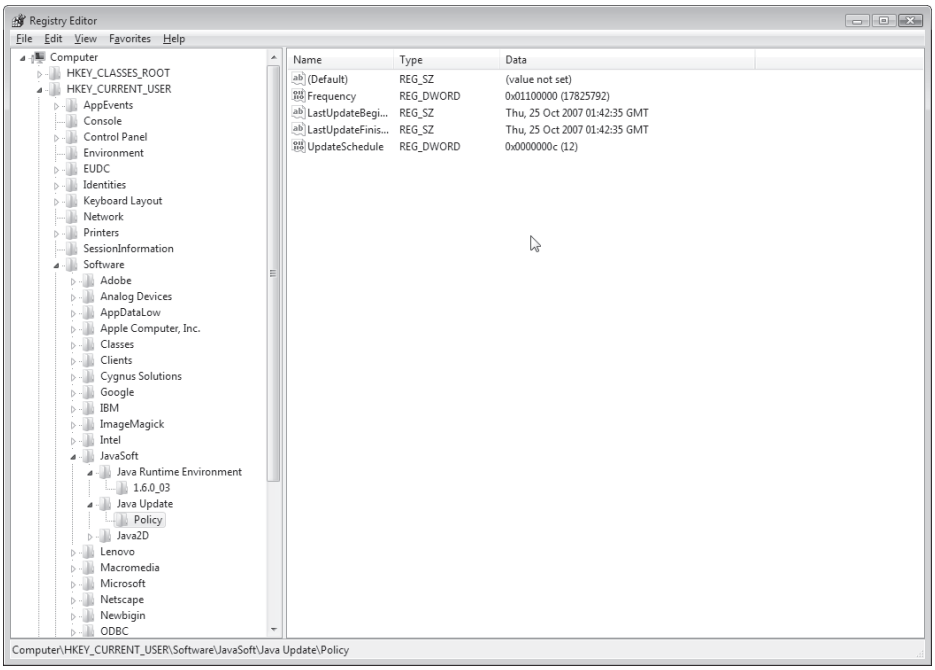


Рис. 12.4. Редактор реестра

Например, в случае, если была установлена версия Java SE 6, ключ

```
HKEY_LOCAL_MACHINE\Software\JavaSoft\Java Runtime Environment
```

будет содержать следующую пару “имя=значение”:

```
CurrentVersion="1.6.0_03"
```

В данном случае значение имеет вид строки. Однако значения также еще могут иметь вид и целых чисел или массивов байтов.

Java-интерфейс для доступа к реестру

Попробуем разработать простой интерфейс для организации доступа к реестру из Java-программы, а также реализовать его с помощью платформенно-ориентированного кода. Этот интерфейс позволяет выполнять только несколько операций с реестром. Для сокращения размера кода в нем не поддерживаются многие важные операции, например, вставка и удаление ключей (при необходимости это можно довольно легко реализовать.)

Даже используя ограниченный набор функций, можно выполнять следующие действия с реестром:

- получать все имена, связанные с ключом;
- считывать значение, соответствующее имени;
- связывать с именем новое значение.

Для выполнения таких действий используется Java-класс, который инкапсулирует 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 = 0x80000000;
   public static final int HKEY_CURRENT_USER = 0x80000001;
   public static final int HKEY_LOCAL_MACHINE = 0x80000002;
   ...
}
```

Метод `names()` возвращает объект `Enumeration`, который содержит все имена, связанные с данным ключом. Их можно получить с помощью методов `hasMoreElements()/nextElement()`. Метод `getValue()` возвращает объект, который может быть строкой, экземпляром класса `Integer` или байтовым массивом. Аналогично: параметр `value` метода `setValue()` также должен соответствовать одному из этих трех типов.

Организация доступа к реестру посредством платформенно-ориентированных методов

Нам необходимо реализовать следующие функциональные возможности:

- получение значения, соответствующего имени;
- связывание с именем нового значения;
- перечисление имен, соответствующих ключу.

К счастью, вы уже знакомы со всеми инструментами, необходимыми для преобразования строк/массивов языков Java и C. Кроме того, рассмотрена генерация и обработка исключений в случае возникновения ошибок.

По сравнению с примерами из предыдущих разделов применение платформенно-ориентированных методов усложняется двумя особенностями. Методы `getValue()` и `setValue()` работают с объектами, которые могут иметь один из следующих типов: `String`, `Integer` или `byte[]`. Кроме того, объект `Enumeration` должен *сохранять информацию о состоянии* между последовательными вызовами методов `hasMoreElements()/nextElement()`.

Давайте сначала рассмотрим метод `getValue()`. Этот метод (код которого приведен в листинге 12.22) выполняет перечисленные ниже действия.

1. Открывает ключ реестра. Для считывания значений API-интерфейс реестра требует, чтобы ключи были открыты.
2. Запрашивает тип и размер значения, ассоциируемого с данным именем.
3. Выполняет считывание данных значения в буфер.
4. Вызывает функцию `NewStringUTF()` для создания новой строки с данными этого значения, если это значение является значением типа `REG_SZ` (строкой).
5. Вызывает конструктор `Integer`, если это значение является значением типа `REG_DWORD` (32-битным целочисленным значением).
6. Если это значение является значением типа `REG_BINARY` (байтовым массивом), вызывает сначала функцию `NewByteArray()` для создания нового байтового массива, а затем — функцию `SetByteArrayRegion()` для копирования в полученный массив данных этого значения.

7. Если же это значение не относится ни к одному из перечисленных типов или если при вызове функции API возникла какая-то ошибка, выдает исключение и освобождает все ресурсы, которые были получены к этому моменту.
8. Закрывает раздел ключа и возвращает созданный объект (`String`, `Integer` или `byte[]`).

Этот пример довольно хорошо иллюстрирует способы генерации Java-объектов различных типов.

В данном платформенно-ориентированном методе совсем несложно организовать возврат универсального значения, поскольку ссылка на объекты типа `jstring`, `jint` или `byte[]` может возвращаться как ссылка `jobject`. Однако метод `setValue()` получает ссылку `Object` и должен определить конкретный тип для сохранения значения в виде строки, целочисленного значения или байтового массива. Этот тип можно определить, запрашивая класс объекта-значения, получая ссылки на классы `java.lang.String`, `java.lang.Integer` и `java.lang.byte[]`, а затем сравнивая их с помощью функции `IsAssignableFrom()`.

Например, для ссылок на классы `class1` и `class2` показанный ниже метод возвратит значение `JNI_TRUE`, если классы `class1` и `class2` совпадают или класс `class1` является подклассом `class2`.

```
(*env)->IsAssignableFrom(env, class1, class2)
```

В любом из этих случаев ссылки на объекты класса `class1` должны приводиться к типу `class2`.

Например, если приведенный ниже метод возвращает значение `true`, это значит, что значение `value` является байтовым массивом.

```
(*env)->IsAssignableFrom(env,
   (*env)->GetObjectClass(env, value),
   (*env)->FindClass(env, "[B"))
```

Теперь давайте рассмотрим метод `setValue()`. Этот метод выполняет следующие действия.

1. Открывает соответствующий ключу раздел реестра для выполнения записи.
2. Узнает тип подлежащего записи значения.
3. Если значение является значением типа `String`, вызывает функцию `GetStringUTFChars()` для получения указателя на символы.
4. Если значение является значением типа `Integer`, вызывает метод `intValue()` для извлечения целого числа, хранящегося в объекте-оболочке.
5. Если значение является значением типа `byte[]`, вызывает метод `GetByteArrayElements()` для получения указателя на байты.
6. Передает данные и длину значения реестру.
7. Закрывает раздел ключа.
8. Освобождает указатель на данные, если речь идет о значении типа `String` или `byte[]`.

И, наконец, рассмотрим платформенно-ориентированные методы для перечисления данных, связанных с ключом. Как видно в листинге 12.21, эти методы входят в состав класса `Win32RegKeyNameEnumeration`. Вначале необходимо открыть раздел, соответствующий ключу. На время существования объекта перечисления нужно хранить дескриптор ключа. Дескриптор имеет тип `DWORD` (32-битное целочисленное значение),

а потому может быть сохранен в поле hkey класса Enumeration. При запуске процесса перечисления это поле инициализируется с помощью метода SetIntField(). В дальнейшем значение поля hkey считывается с помощью метода GetIntField().

В данном примере в объекте Enumeration также сохраняются три других элемента данных. При запуске процесса перечисления можно вычислить длину самого длинного имени. Эта информация помещается в поля count и maxsize объекта Enumeration и используется для создания символьного массива на языке C. Наконец, поле index инициализируется значением -1 до начала процесса перечисления, в нем устанавливается значение 0 при инициализации других полей экземпляра и оно увеличивается на единицу на каждом шаге перечисления.

Давайте посмотрим, что именно делают платформенно-ориентированные методы, поддерживающие перечисление. Метод hasMoreElements() является довольно простым и выполняет следующие действия.

1. Извлекает значения полей index и count.
2. Если значением поля index является -1, вызывает функцию startNameEnumeration(), которая открывает соответствующий раздел ключа, запрашивает значение счетчика и максимальную длину, а затем инициализирует поля hkey, count, maxsize и index.
3. В случае если значение поля index меньше значения поля count, возвращает значение JNI_TRUE, а в противном случае — значение JNI_FALSE.

Метод nextElement() является более сложным и выполняет перечисленные ниже действия.

1. Извлекает значения полей index и count.
2. Если значение поля index равно -1, вызывает функцию startNameEnumeration(), которая открывает соответствующий раздел ключа, запрашивает значение счетчика и максимальную длину, а затем инициализирует поля hkey, count, maxsize и index.
3. Если значение поля index равно значению поля count, выдает исключение NoSuchElementException.
4. Считывает следующее имя из реестра.
5. Увеличивает значение поля index.
6. Если значение поля index снова оказывается равным значению поля count, закрывает раздел ключа.

Перед компиляцией следует обязательно не забыть запустить утилиту javah как для Win32RegKey, так и для Win32RegKeyNameEnumeration. Для компилятора Microsoft вся необходимая командная строка будет выглядеть так:

```
cl -I jdk\include -I jdk\include\win32 -LD Win32RegKey.c
   advapi32.lib -FeWin32RegKey.dll
```

Для Cygwin требуемая команда будет выглядеть следующим образом:

```
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
```

Поскольку API-интерфейс реестра рассчитан исключительно на Windows, в других операционных системах данная программа работать не будет.

В листинге 12.23 приведен код программы, позволяющей протестировать созданные нами новые функции для работы с реестром. Для проведения такого тестирования мы сначала добавляем три пары "имя=значение" (строку, целое число и байтовый массив) в следующий ключ:

```
HKEY_CURRENT_USER\Software\JavaSoft\Java Runtime Environment
```

Затем мы перечисляем все имена этого ключа и извлекаем их значения. Вывод этой программы должен выглядеть следующим образом:

```
Default user=Harry Hacker
Lucky number=13
Small primes=2 3 5 7 11 13
```

Хотя добавление таких пар имен и значений в данный ключ, пожалуй, никак не может навредить, при желании после выполнения данной программы их можно легко удалить с помощью редактора реестра.

Листинг 12.21. Содержимое файла `Win32RegKey.java`

```
1.  import java.util.*;
2.
3.  /**
4.   * Объект Win32RegKey может применяться для
5.   * извлечения и установки значений,
6.   * соответствующих ключу реестра Windows.
7.   * @version 1.00 1997-07-01
8.   * @author Cay Horstmann
9.   */
10. public class Win32RegKey
11. {
12.    /**
13.     * Создание объекта, представляющего ключ реестра.
14.     * @param theRoot Одно из следующих значений:
15.     * HKEY_CLASSES_ROOT, HKEY_CURRENT_USER,
16.     * HKEY_LOCAL_MACHINE, HKEY_USERS, HKEY_CURRENT_CONFIG,
17.     * HKEY_DYN_DATA.
18.     * @param thePath Путь к ключу реестра.
19.     */
20.    public Win32RegKey(int theRoot, String thePath)
21.    {
22.       root = theRoot;
23.       path = thePath;
24.    }
25.
26.    /**
27.     * Перечисление всех имен для пути, указанного в объекте.
28.     * @return Объект Enumeration, содержащий информацию об именах
29.     */
30.    public Enumeration<String> names()
31.    {
32.       return new Win32RegKeyNameEnumeration(root, path);
33.    }
34.
35.    /**
36.     * Получение значения, связанного с именем.
37.     * @param name Имя
38.     * @return Значение, соответствующее указанному имени
39.     */
```

```
40.    public native Object getValue(String name);
41.
42.    /**
43.     * Связывание значения с именем.
44.     * @param name Имя
45.     * @param value Новое значение
46.     */
47.    public native void setValue(String name, Object value);
48.
49.    public static final int HKEY_CLASSES_ROOT = 0x80000000;
50.    public static final int HKEY_CURRENT_USER = 0x80000001;
51.    public static final int HKEY_LOCAL_MACHINE = 0x80000002;
52.    public static final int HKEY_USERS = 0x80000003;
53.    public static final int HKEY_CURRENT_CONFIG = 0x80000005;
54.    public static final int HKEY_DYN_DATA = 0x80000006;
55.
56.    private int root;
57.    private String path;
58.
59.    static
60.    {
61.       System.loadLibrary("Win32RegKey");
62.    }
63. }
64.
65. class Win32RegKeyNameEnumeration implements Enumeration<String>
66. {
67.    Win32RegKeyNameEnumeration(int theRoot, String thePath)
68.    {
69.       root = theRoot;
70.       path = thePath;
71.    }
72.
73.    public native String nextElement();
74.    public native boolean hasMoreElements();
75.
76.    private int root;
77.    private String path;
78.    private int index = -1;
79.    private int hkey = 0;
80.    private int maxsize;
81.    private int count;
82. }
83.
84. class Win32RegKeyException extends RuntimeException
85. {
86.    public Win32RegKeyException()
87.    {
88.    }
89.
90.   public Win32RegKeyException(String why)
91.   {
92.     super(why);
93.   }
94. }
```

Листинг 12.22. Содержимое файла `Win32RegKey.c`

```
/**
 * @version 1.00 1997-07-01
 * @author Cay Horstmann
 */

#include "Win32RegKey.h"
#include "Win32RegKeyNameEnumeration.h"
#include <string.h>
#include <stdlib.h>
#include <windows.h>

JNIEXPORT jobject JNICALL Java_Win32RegKey_getValue(JNIEnv* env,
    jobject this_obj, jobject name)
{
   const char* cname;
   jstring path;
   const char* cpath;
   HKEY hkey;
   DWORD type;
   DWORD size;
   jclass this_class;
   jfieldID id_root;
   jfieldID id_path;
   HKEY root;
   jobject ret;
   char* cret;

   /* Получение класса. */
   this_class = (*env)->GetObjectClass(env, this_obj);

   /* Получение идентификаторов полей. */
   id_root = (*env)->GetFieldID(env, this_class, "root", "I");
   id_path = (*env)->GetFieldID(env, this_class, "path",
     "Ljava/lang/String;");

   /* Получение полей. */
   root = (HKEY) (*env)->GetIntField(env, this_obj, id_root);
   path = (jstring)(*env)->GetObjectField(env, this_obj, id_path);
   cpath = (*env)->GetStringUTFChars(env, path, NULL);

   /* Открытие раздела, соответствующего ключу реестра. */
   if (RegOpenKeyEx(root, cpath, 0, KEY_READ, &hkey) !=
      ERROR_SUCCESS)
   {
      (*env)->ThrowNew(env,
        (*env)->FindClass(env, "Win32RegKeyException"),
        "Open key failed");
        /* Не удалось открыть ключ */
      (*env)->ReleaseStringUTFChars(env, path, cpath);
      return NULL;
   }

   (*env)->ReleaseStringUTFChars(env, path, cpath);
   cname = (*env)->GetStringUTFChars(env, name, NULL);

```

```
56.    /* Определение типа и размера значения. */
57.    if (RegQueryValueEx(hkey, cname, NULL, &type, NULL, &size)
58.       != ERROR_SUCCESS)
59.    {
60.       (*env)->ThrowNew(env,
61.          (*env)->FindClass(env, "Win32RegKeyException"),
62.          "Query value key failed");
63.          /* Запрос значения ключа не удался */
64.       RegCloseKey(hkey);
65.       (*env)->ReleaseStringUTFChars(env, name, cname);
66.       return NULL;
67.    }
68.
69.    /* Выделение памяти для размещения значения. */
70.    cret = (char*)malloc(size);
71.
72.    /* Чтение значения. */
73.    if (RegQueryValueEx(hkey, cname, NULL, &type, cret, &size)
74.       != ERROR_SUCCESS)
75.    {
76.       (*env)->ThrowNew(env,
77.          (*env)->FindClass(env, "Win32RegKeyException"),
78.          "Query value key failed");
79.          /* Запрос значения ключа не удался */
80.       free(cret);
81.       RegCloseKey(hkey);
82.       (*env)->ReleaseStringUTFChars(env, name, cname);
83.       return NULL;
84.    }
85.
86.    /* В зависимости от типа, значение помещается в строку,
87.       целочисленную переменную или байтовый массив. */
88.    if (type == REG_SZ)
89.    {
90.       ret = (*env)->NewStringUTF(env, cret);
91.    }
92.    else if (type == REG_DWORD)
93.    {
94.       jclass class_Integer = (*env)->FindClass(env,
95.          "java/lang/Integer");
96.       /* Получение идентификатора конструктора. */
97.       jmethodID id_Integer = (*env)->GetMethodID(env,
98.          class_Integer, "<init>", "(I)V");
99.       int value = *(int*) cret;
100.      /* Вызов конструктора. */
101.      ret = (*env)->NewObject(env, class_Integer,
102.         id_Integer, value);
103.   }
104.   else if (type == REG_BINARY)
105.   {
106.      ret = (*env)->NewByteArray(env, size);
107.      (*env)->SetByteArrayRegion(env, (jarray) ret, 0,
108.         size, cret);
109.   }
110.   else
111.   {
112.      (*env)->ThrowNew(env,
```

```
113.         (*env)->FindClass(env, "Win32RegKeyException"),
114.         "Unsupported value type");
115.         /* Неподдерживаемый тип значения */
116.       ret = NULL;
117.    }
118.
119.    free(cret);
120.    RegCloseKey(hkey);
121.    (*env)->ReleaseStringUTFChars(env, name, cname);
122.
123.    return ret;
124. }
125.
126. JNIEXPORT void JNICALL Java_Win32RegKey_setValue(JNIEnv* env,
127.    jobject this_obj, jstring name, jobject value)
128. {
129.    const char* cname;
130.    jstring path;
131.    const char* cpath;
132.    HKEY hkey;
133.    DWORD type;
134.    DWORD size;
135.    jclass this_class;
136.    jclass class_value;
137.    jclass class_Integer;
138.    jfieldID id_root;
139.    jfieldID id_path;
140.    HKEY root;
141.    const char* cvalue;
142.    int ivalue;
143.
144.    /* Получение класса. */
145.    this_class = (*env)->GetObjectClass(env, this_obj);
146.
147.    /* Получение идентификаторов полей. */
148.    id_root = (*env)->GetFieldID(env, this_class, "root", "I");
149.    id_path = (*env)->GetFieldID(env, this_class, "path",
150.       "Ljava/lang/String;");
151.
152.    /* Получение полей. */
153.    root = (HKEY)(*env)->GetIntField(env, this_obj, id_root);
154.    path = (jstring)(*env)->GetObjectField(env, this_obj, id_path);
155.    cpath = (*env)->GetStringUTFChars(env, path, NULL);
156.
157.    /* Открытие раздела, соответствующего ключу реестра. */
158.    if (RegOpenKeyEx(root, cpath, 0, KEY_WRITE, &hkey) !=
159.       ERROR_SUCCESS)
160.    {
161.       (*env)->ThrowNew(env,
162.          (*env)->FindClass(env, "Win32RegKeyException"),
163.          "Open key failed");
164.        /* Не удалось открыть ключ */
165.       (*env)->ReleaseStringUTFChars(env, path, cpath);
166.       return;
167.    }
168.
```

```
   (*env)->ReleaseStringUTFChars(env, path, cpath);
   cname = (*env)->GetStringUTFChars(env, name, NULL);

   class_value = (*env)->GetObjectClass(env, value);
   class_Integer = (*env)->FindClass(env, "java/lang/Integer");
   /* Определение типа значения. */
   if ((*env)->IsAssignableFrom(env, class_value,
      (*env)->FindClass(env, "java/lang/String")))
   {
      /* Строка - получение указателя. */
      cvalue = (*env)->GetStringUTFChars(env, (jstring) value,
        NULL);
      type = REG_SZ;
      size = (*env)->GetStringLength(env, (jstring) value) + 1;
   }
   else if ((*env)->IsAssignableFrom(env, class_value,
      class_Integer))
   {
      /* Целое число - вызов intValue для получения значения */
      jmethodID id_intValue = (*env)->GetMethodID(env,
        class_Integer, "intValue", "()I");
      ivalue = (*env)->CallIntMethod(env, value, id_intValue);
      type = REG_DWORD;
      cvalue = (char*)&ivalue;
      size = 4;
   }
   else if ((*env)->IsAssignableFrom(env, class_value,
     (*env)->FindClass(env, "[B")))
   {
      /* Байтовый массив - получение указателя. */
      type = REG_BINARY;
      cvalue = (char*)(*env)->GetByteArrayElements(env,
        (jarray) value, NULL);
      size = (*env)->GetArrayLength(env, (jarray) value);
   }
   else
   {
      /* Данный тип не поддерживается. */
      (*env)->ThrowNew(env,
        (*env)->FindClass(env, "Win32RegKeyException"),
        "Unsupported value type");
        /* Не поддерживаемый тип значения */
      RegCloseKey(hkey);
      (*env)->ReleaseStringUTFChars(env, name, cname);
      return;
   }

   /* Установка значения. */
   if (RegSetValueEx(hkey, cname, 0, type, cvalue, size)
      != ERROR_SUCCESS)
   {
      (*env)->ThrowNew(env,
        (*env)->FindClass(env, "Win32RegKeyException"),
        "Set value failed");
       /* Установка значения не удалась */
   }

```

```
226.    RegCloseKey(hkey);
227.    (*env)->ReleaseStringUTFChars(env, name, cname);
228.
229.    /* Если значение представляло собой строку или байтовый массив,
230.       освобождается занимаемая им память. */
231.    if (type == REG_SZ)
232.    {
233.       (*env)->ReleaseStringUTFChars(env, (jstring) value, cvalue);
234.    }
235.    else if (type == REG_BINARY)
236.    {
237.       (*env)->ReleaseByteArrayElements(env, (jarray) value,
238.          (jbyte*) cvalue, 0);
239.    }
240. }
241.
242. /* Вспомогательная функция для перечисления имен. */
243. static int startNameEnumeration(JNIEnv* env, jobject this_obj,
244.    jclass this_class)
245. {
246.    jfieldID id_index;
247.    jfieldID id_count;
248.    jfieldID id_root;
249.    jfieldID id_path;
250.    jfieldID id_hkey;
251.    jfieldID id_maxsize;
252.
253.    HKEY root;
254.    jstring path;
255.    const char* cpath;
256.    HKEY hkey;
257.    DWORD maxsize = 0;
258.    DWORD count = 0;
259.
260.    /* Получение идентификаторов полей. */
261.    id_root = (*env)->GetFieldID(env, this_class, "root", "I");
262.    id_path = (*env)->GetFieldID(env, this_class, "path",
263.       "Ljava/lang/String;");
264.    id_hkey = (*env)->GetFieldID(env, this_class, "hkey", "I");
265.    id_maxsize = (*env)->GetFieldID(env, this_class,
266.       "maxsize", "I");
267.    id_index = (*env)->GetFieldID(env, this_class, "index", "I");
268.    id_count = (*env)->GetFieldID(env, this_class, "count", "I");
269.
270.    /* Получение значений полей. */
271.    root = (HKEY)(*env)->GetIntField(env, this_obj, id_root);
272.    path = (jstring)(*env)->GetObjectField(env, this_obj, id_path);
273.    cpath = (*env)->GetStringUTFChars(env, path, NULL);
274.
275.    /* Открытие раздела, соответствующего ключу реестра. */
276.    if (RegOpenKeyEx(root, cpath, 0, KEY_READ, &hkey) !=
277.       ERROR_SUCCESS)
278.    {
279.       (*env)->ThrowNew(env,
280.          (*env)->FindClass(env, "Win32RegKeyException"),
281.          "Open key failed");
282.          /* Не удалось открыть ключ */
```

```
283.      (*env)->ReleaseStringUTFChars(env, path, cpath);
284.      return -1;
285.    }
286.    (*env)->ReleaseStringUTFChars(env, path, cpath);
287.
288.    /* Запрос счетчика имен и максимальной длины имени. */
289.    if (RegQueryInfoKey(hkey, NULL, NULL, NULL, NULL, NULL, NULL,
290.         &count, &maxsize, NULL, NULL, NULL) != ERROR_SUCCESS)
291.    {
292.      (*env)->ThrowNew(env,
293.        (*env)->FindClass(env, "Win32RegKeyException"),
294.        "Query info key failed");
295.        /* Запрос информации о ключе не удался */
296.      RegCloseKey(hkey);
297.      return -1;
298.    }
299.
300.    /* Установка значений полей. */
301.    (*env)->SetIntField(env, this_obj, id_hkey, (DWORD) hkey);
302.    (*env)->SetIntField(env, this_obj, id_maxsize, maxsize + 1);
303.    (*env)->SetIntField(env, this_obj, id_index, 0);
304.    (*env)->SetIntField(env, this_obj, id_count, count);
305.    return count;
306. }
307.
308. JNIEXPORT jboolean JNICALL
309.    Java_Win32RegKeyNameEnumeration_hasMoreElements
310.    (JNIEnv* env, jobject this_obj)
311. {  jclass this_class;
312.    jfieldID id_index;
313.    jfieldID id_count;
314.    int index;
315.    int count;
316.    /* Получение класса. */
317.    this_class = (*env)->GetObjectClass(env, this_obj);
318.
319.    /* Получение идентификаторов полей. */
320.    id_index = (*env)->GetFieldID(env, this_class, "index", "I");
321.    id_count = (*env)->GetFieldID(env, this_class, "count", "I");
322.
323.    index = (*env)->GetIntField(env, this_obj, id_index);
324.    if (index == -1)
325.    {
326.      count = startNameEnumeration(env, this_obj, this_class);
327.      index = 0;
328.    }
329.    else
330.      count = (*env)->GetIntField(env, this_obj, id_count);
331.    return index < count;
332. }
333.
334. JNIEXPORT jobject JNICALL
335.     Java_Win32RegKeyNameEnumeration_nextElement
336.     (JNIEnv* env, jobject this_obj)
337. {
338.    jclass this_class;
339.    jfieldID id_index;
```

```
340.    jfieldID id_hkey;
341.    jfieldID id_count;
342.    jfieldID id_maxsize;
343.
344.    HKEY hkey;
345.    int index;
346.    int count;
347.    DWORD maxsize;
348.
349.    char* cret;
350.    jstring ret;
351.
352.    /* Получение класса. */
353.    this_class = (*env)->GetObjectClass(env, this_obj);
354.
355.    /* Получение идентификаторов полей. */
356.    id_index = (*env)->GetFieldID(env, this_class, "index", "I");
357.    id_count = (*env)->GetFieldID(env, this_class, "count", "I");
358.    id_hkey = (*env)->GetFieldID(env, this_class, "hkey", "I");
359.    id_maxsize = (*env)->GetFieldID(env, this_class,
360.       "maxsize", "I");
361.
362.    index = (*env)->GetIntField(env, this_obj, id_index);
363.    if (index == -1)
364.    {
365.       count = startNameEnumeration(env, this_obj, this_class);
366.       index = 0;
367.    }
368.    else
369.       count = (*env)->GetIntField(env, this_obj, id_count);
370.
371.    if (index >= count) /* Содержимое исчерпано. */
372.    {
373.       (*env)->ThrowNew(env, (*env)->FindClass(env,
374.          "java/util/NoSuchElementException"),
375.          "past end of enumeration");
376.          /* конец перечисления */
377.       return NULL;
378.    }
379.
380.    maxsize = (*env)->GetIntField(env, this_obj, id_maxsize);
381.    hkey = (HKEY)(*env)->GetIntField(env, this_obj, id_hkey);
382.    cret = (char*)malloc(maxsize);
383.
384.    /* Поиск следующего имени. */
385.    if (RegEnumValue(hkey, index, cret, &maxsize,
386.       NULL, NULL, NULL, NULL)!= ERROR_SUCCESS)
387.    {
388.       (*env)->ThrowNew(env,
389.          (*env)->FindClass(env, "Win32RegKeyException"),
390.          "Enum value failed");
391.          /* Не удалось перечислить значение */
392.       free(cret);
393.       RegCloseKey(hkey);
394.       (*env)->SetIntField(env, this_obj, id_index, count);
395.       return NULL;
396.    }
```

```
   ret = (*env)->NewStringUTF(env, cret);
   free(cret);

   /* Увеличение индекса. */
   index++;
   (*env)->SetIntField(env, this_obj, id_index, index);

   if (index == count) /* at end */
   {
     RegCloseKey(hkey);
   }

   return ret;
}
```

Листинг 12.23. Содержимое файла `Win32RegKeyTest.java`

```
import java.util.*;

/**
 * @version 1.02 2007-10-26
 * @author Cay Horstmann
 */
public class Win32RegKeyTest
{
   public static void main(String[] args)
   {
      Win32RegKey key = new Win32RegKey(
         Win32RegKey.HKEY_CURRENT_USER,
            "Software\\JavaSoft\\Java Runtime Environment");

      key.setValue("Default user", "Harry Hacker");
      key.setValue("Lucky number", new Integer(13));
      key.setValue("Small primes", new byte[] { 2, 3, 5, 7, 11 });

      Enumeration<String> e = key.names();

      while (e.hasMoreElements())
      {
         String name = e.nextElement();
         System.out.print(name + "=");

         Object value = key.getValue(name);

         if (value instanceof byte[])
            for (byte b : (byte[]) value)
               System.out.print((b & 0xFF) + " ");
         else
            System.out.print(value);

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

API Функции определения типа

- `jboolean IsAssignableFrom(JNIEnv *env, jclass cl1, jclass cl2)`
  В случае если объекты первого класса (`cl1`) могут быть присвоены объектам второго класса (`cl2`), возвращает значение `JNI_TRUE`, а если нет — значение `JNI_FALSE`. Присваиваться они могут тогда, когда классы `cl1` и `cl2` являются одинаковыми, когда класс `cl1` является подклассом класса `cl2` или когда класс `cl2` представляет интерфейс, который реализуется классом `cl1` или каким-то из его суперклассов.
- `jclass GetSuperClass(JNIEnv *env, jclass cl)`
  Возвращает суперкласс указанного класса. Если `cl` представляет класс `Object` или интерфейс, тогда возвращает значение `NULL`.

Вот и подошел к концу второй том этой книги, завершающий длинное путешествие читателя по миру Java, во время которого у него была возможность познакомиться со многими сложными API-интерфейсами. Начиналось все с рассмотрения таких необходимых каждому Java-программисту тем, как потоки, XML, сетевые подключения, базы данных и интернационализация. Далее шли три долгих главы по графике и программированию GUI. А напоследок вниманию читателя были предложены несколько очень технических глав, посвященных вопросам безопасности, удаленным методам, способам обработки аннотаций и платформенно-ориентированным методам. Мы очень надеемся, что читателю понравилась эта "экскурсия" по увлекательному миру API-интерфейсов Java и что теперь он сможет применять все полученные знания в своих проектах.

Предметный указатель

A

Active Server Pages (ASP), 218
Advanced Encryption Standard (AES), 811
Application Programming Interface (API), 20
  вызова, 950
  компилятора, 877

B

Basic Encoding Rules (BER), 795

C

CDATA-раздел, 108
Common Gateway Interface (CGI), 218
Common Object Request Broker Architecture (CORBA), 826

D

Data Encryption Standard (DES), 810
Digital Signature Algorithm (DSA), 793
Distinguished Encoding Rules (DER), 795
Document Type Definition (DTD), 106; 121
DOM (Document Object Model), 109
DOM-анализатор, 109

I

IBM Tivoli Directory Server, 289
Interface Definition Language (IDL), 826

J

JAR-архив, 675
JAR-файл, 237
Java 2D API, 516
Java API for XML Processing (JAXP), 109
Java Authentication and Authorization Service (JAAS), 771
JavaBeans, 670
Java DataBase Connectivity (JDBC), 228; 236
Java Naming and Directory Interface (JNDI), 228; 287
JavaServer Faces, 218

L

LDAP-сервер, 289
Lightweight Directory Access Protocol (LDAP), 228; 288

M

Microsoft Active Directory, 289

N

NetBeans, 670
Novell eDirectory, 289

O

ODBC, 228
OpenLDAP, 289

Q

Query by example (QBE), 234

R

Remote Method Invocation (RMI), 826

S

SAX-анализатор, 109; 152
Simple API for XML (SAX), 109
Simple Mail Transport Protocol (SMTP), 202
Simple Object Access Protocol (SOAP), 826
Standard Generilized Markup Language (SGML), 105
Structured Query Language (SQL), 228; 231
Sun Java System Directory Server, 289
Swing, 356

T

Telnet, 182

U

URI (uniform resource identificator)
  абсолютный, 207
  непрозрачный, 207
  относительный, 207
    разрешение относительного URI, 208
URL (uniform resource locator), 207
  базы данных, 237
URN (uniform resource name), 207

W

Web Services Description Language (WSDL), 826

X

XML, 104
  -схема, 128
XPath, 144
XSLT (Extensible Stylesheet Language Transformations), 161
XSLT-преобразования, 171

А

Алгоритм
  AES (расширенный стандарт шифрования), 811
  DES (стандарт шифрования данных), 810
  DSA (создания цифровых подписей), 793
  рисования, 517
Альфа-канал, 550
Анализатор (parser), 109
Аннотация, 887
Архив
  JAR, 675
  ZIP, 49
Атрибут
  документа, 614
  задания печати, 615
  запроса печати, 614
  категория атрибутов, 617
  службы печати, 615
Аутентификация, 771
  задача аутентификации, 799
  на основе ролей, 777
Аффинное преобразование, 545

Б

База данных, 228
Безопасность, 738
Бизнес-логика, 231
Буфер, 87
  автоматическая очистка, 30
  обмена, 621

В

Верификатор, 748
Верификация байтового кода, 748
Визуальное представление, 231
Вставка, 621
Выборочные значения, 575
Выделение контуров, 584
Выражение
  регулярное, 91
    синтаксис, 93
Вырезание, 621

Г

Градиентное заполнение, 541

Д

Дайджест сообщения, 787
Диспетчер защиты, 753
Документ
  генерация XML-документа, 160
  определение типа документа, 106
Домен защиты, 756
Драйвер, 228
  типа 1, 229
  типа 2, 229
  типа 3, 230
  типа 4, 230
Дуга, 519

З

Загрузчик классов, 739
Запись, 232
Запрос
  к базе данных, 251
  по образцу, 234
Защита, 738

И

Изображение
  прозрачное, 549
Имя,
  квалифицированное, 151
  локальное, 151
  пространство имен, 150
Индикатор хода процесса, 476
Инструкция обработки, 108
Интернационализация, 306
Интерполяция, 582
Интерфейс
  ActivationSystem, 853
  Appendable, 25
  Attribute, 303; 615
  AttributeSet, 621
  BeanInfo, 694; 695; 700; 708
  CachedRowSet, 270; 272
  Callback, 778
  CallbackHandler, 785
  CellEditor, 406
  CharacterData, 120
  CharSequence, 25; 26
  ClipboardOwner, 623; 627
  Closeable, 25
  Composite, 552
  Connection, 242; 243; 259; 268
  DatabaseMetaData, 269; 281
  DataInput, 42
  DataOutput, 41
  DirContext, 302
  DocAttribute, 615
  DocAttributeSet, 616
  DocPrintJob, 613; 621
  Document, 119; 163
  EntityResolver, 127
  ExceptionListener, 734
  Externalizable, 62

FilteredRowSet, 270
FlavorListener, 629
Flushable, 25
HyperlinkListener, 472; 475
Invocable, 870; 872
JdbcRowSet, 270
JoinRowSet, 270
ListCellRenderer, 373
ListModel, 367
ListSelectionListener, 362
ListSelectionModel, 393
LoginModule, 786
MutableTreeNode, 415
NamedNodeMap, 120
NodeList, 120
Pageable, 599
PreparedStatement, 259
Printable, 591
PrintJobAttribute, 615
PrintJobAttributeSet, 616
PrintRequestAttribute, 615
PrintRequestAttributeSet, 616
PrintService, 613; 621
PrintServiceAttribute, 615
PrintServiceAttributeSet, 616
PrivilegedAction, 777
PrivilegedExceptionAction, 777
Readable, 24; 25
ReadableByteChannel, 196
Remote, 829
Result, 175
ResultSet, 244; 247; 260; 261; 263; 268; 282
ResultSetMetaData, 282
RowSet, 270; 272
Savepoint, 285
Serializable, 56
Shape, 533
SupportedValuesAttribute, 615
TableCellEditor, 398; 406
TableCellRenderer, 405
TableColumnModel, 393
TableModel, 391
Transferable, 622; 627; 629
TreeCellRenderer, 427
TreeModel, 415; 435
TreeModelListener, 435; 441
TreeNode, 415; 422
TreeSelectionListener, 428; 433
VetoableChangeListener, 504; 514; 692
WebRowSet, 270
WritableByteChannel, 196
XMLReader, 180
многодокументный, 498
поставщика услуг, 565

Исключение
AlreadyBoundException, 834
IllegalArgumentException, 946
IOException, 185; 472
NotBoundException, 834
NullPointerException, 946
PropertyVetoException, 505; 514; 693
RemoteException, 829
SAXParseException, 127
UnknownHostException, 185
UnsupportedFlavorException, 627

K

Класс
Activatable, 852
ActivationGroup, 853
ActivationGroupDesc, 852
AffineTransform, 546
AlphaComposite, 552; 557
Applet, 311; 354
Arc2D, 518; 519
Area, 533
Attributes, 156; 302
AttributesImpl, 180
Banner, 600
BasicAttributes, 303
BasicStroke, 540
BeanDescriptor, 708
Bindings, 869
Book, 609
Buffer, 86; 87; 88
BufferedInputStream, 28
BufferedOutputStream, 29
BufferedReader, 32
ByteBuffer, 40; 86
ByteLookupTable, 589
CharBuffer, 24; 40; 88
Channels, 202
Charset, 37; 40
Cipher, 815
CipherInputStream, 817
CipherOutputStream, 817
Class, 748; 757
ClassLoader, 742; 748
Clipboard, 622; 626; 628
CodeSource, 757
Color, 541; 581
ColorModel, 576; 580
Compilable, 872
CompilationTask, 878
CompiledScript, 872
ContentHandler, 156
Context, 302; 833
ConvolveOp, 590

Copies, 617
CubicCurve2D, 518
Currency, 317
DatabaseMetaData, 245
DataFlavor, 623; 628
DataInput, 42
DataInputStream, 26; 32; 42
DataOutput, 43
DataOutputStream, 32
DateFormat, 324
DefaultCellEditor, 406; 418
DefaultListModel, 368
DefaultMutableTreeNode, 415; 427
DefaultPersistenceDelegate, 735
DefaultTreeCellRenderer, 427
DefaultTreeModel, 415; 423
DocumentBuilder, 119; 126; 160; 163
DocumentBuilderFactory, 127
DOMResult, 175; 180
DOMSource, 164
Element, 119; 163
Ellipse2D, 518; 519
Expression, 735
FeatureDescriptor, 696
File, 73; 77
FileChannel, 85; 91
FileInputStream, 26; 28; 85
FileLock, 91
FilenameFilter, 76
FileOutputStream, 26; 28; 85
FilterInputStream, 27
FilterOutputStream, 27
Format, 334
GeneralPath, 518; 519; 532
GradientPaint, 541; 542
Graphics, 516; 549
Graphics2D, 516; 518
HashAttributeSet, 616
HashDocAttributeSet, 616
HashPrintJobAttributeSet, 616
HashPrintRequestAttributeSet, 616
HashPrintServiceAttributeSet, 616
HyperlinkEvent, 472; 476
IIOImage, 574
IIOServiceProvider, 573
ImageReader, 572
ImageReaderWriterSpi, 573
ImageWriter, 574
IndexedPropertyChangeEvent, 692
IndexedPropertyDescriptor, 697
InetSocketAddress, 202
InputSource, 127
InputStream, 20; 21; 185
Introspector, 695
JColorChooser, 643
JComponent, 415; 514; 693
JDesktopPane, 512
JEditorPane, 469; 471; 475
JFileChooser, 643
JInternalFrame, 513
JList, 366; 368; 373; 643
JProgressBar, 476
JSplitPane, 492
JTabbedPane, 497
JTable, 374; 380; 405; 445; 446; 644
JTextComponent, 643
JTree, 407; 414; 433; 644
Kernel, 590
LabeledPoint, 66
Line2D, 518; 519
Locale, 310
MarshalledObject, 852
Matcher, 100
MessageDigest, 788; 792
MessageFormat, 333
NameCallback, 778; 785
NameClassPair, 834
Naming, 834
NumberFormat, 316
ObjectInputStream, 61
ObjectOutputStream, 56; 61
OutputStream, 20; 22
OutputStreamWriter, 29
PageFormat, 593; 599
PasswordCallback, 778; 786
Pattern, 99
Permission, 771
PersistenceDelegate, 734
Point2D, 519
Principal, 777
PrinterJob, 593; 598; 609
PrintPreviewDialog, 601
PrintServiceLookup, 613
PrintWriter, 30; 31; 220
ProgressMonitor, 476; 487
ProgressMonitorInputStream, 476; 483; 488
Properties, 104
PropertyChangeEvent, 514; 691
PropertyChangeSupport, 691
PropertyDescriptor, 696
ProtectionDomain, 757
PushbackInputStream, 27; 29
QuadCurve2D, 518
RandomAccessFile, 43; 44; 49; 85
Raster, 580
Reader, 24

Rectangle2D, 518; 519
RenderingHints, 564
RescaleOp, 589
ResourceBundle, 341
RoundRectangle2D, 518; 519
SAXParser, 156
SAXParserFactory, 155
SAXResult, 175
SAXSource, 180
Scanner, 196
ScriptContext, 870
ScriptEngine, 869; 870
ScriptEngineManager, 869
SecretKeyFactory, 816
SecretKeySpec, 816
SecurityManager, 757
ShapeMaker, 524
SimpleBeanInfo, 696
SimpleDoc, 613
SimpleJavaFileObject, 882
SimulatedActivity, 477
Socket, 185; 187
SocketChannel, 196; 202
Statement, 242; 243; 735
StreamPrintServiceFactory, 614
StreamResult, 164; 175
StreamSource, 179
StringBuffer, 87
StringSelection, 623
Subject, 776
TableColumn, 383; 393; 405; 460
TextLayout, 549
TexturePaint, 541
Thread, 748
TimeZone, 325
Toolkit, 626
TransferHandler, 643
Transformer, 163
TransformerFactory, 163; 179
TreeModelEvent, 441
TreePath, 422
TreeSelectionEvent, 434
URL, 207; 216
URLConnection, 207; 208; 216; 222
URLDecoder, 225
URLEncoder, 225
VetoableChangeSupport, 692
WritableRaster, 580
Writer, 24
XMLDecoder, 734
XMLEncoder, 733
XPath, 149
XPathFactory, 149
ZipEntry, 54
ZipFile, 55
ZipInputStream, 54
передачи данных, 623
системный, 757
Ключ
открытый, 793
секретный, 793
Кодировка символов
ISO-8859-1, 38
ISO 8859-1, 29
ISO8859-13, 38
ISO8859-15, 38
ISO8859-2, 38
ISO8859-4, 38
ISO8859-5, 38
ISO8859-7, 38
ISO8859-9, 38
Unicode, 29
US-ASCII, 38
UTF-16, 29; 38; 42
UTF-16BE, 38
UTF-16LE, 38
UTF-8, 38; 42
windows-1250, 38
windows-1251, 38
windows-1252, 39
windows-1253, 39
windows-1254, 39
windows-1257, 39
Комментарии, 109
Компонент
JList, 356
JTable, 373
информационный класс компонента, 682
Компоновка, 550
Контроль доступа, 738
Координаты
аппаратные, 543
пользовательские, 543
Корневой узел, 407
Корневой элемент, 107
Косое соединение, 534
Кривая
второго порядка, 522
третьего порядка, 522
Кэшируемый набор строк, 270

Л

Лес, 407
Лист, 407
Локальное имя, 151

M

Маршализация параметров, 827
Масштабирование, 543
Метод
  abort, 786
  absolute, 268
  acceptChanges, 273
  add, 532
  addAttribute, 180
  addBatch, 285
  addCellEditorListener, 406
  addChangeListener, 498
  addElement, 368
  addFlavorListener, 629
  addHyperlinkListener, 475
  addListSelectionListener, 362
  addPropertyChangeListener, 691; 693; 709
  addTab, 497
  addTreeModelListener, 441
  addVetoableChangeListener, 514; 692; 693
  afterLast, 268
  aliases, 38
  append, 532; 609
  appendChild, 163
  applyPattern, 333
  available, 21; 22
  availableCharsets, 38
  beforeFirst, 268
  bind, 834
  breadthFirstEnumeration, 423; 427
  cancelCellEditing, 406
  cancelRowUpdates, 269
  canInsertImage, 574
  checkError, 30; 32
  children, 422
  clear, 88
  clearParameters, 259
  clip, 548; 549
  close, 22
  closePath, 523; 532
  commit, 285; 786
  compare, 331
  connect, 209; 217
  createElement, 161; 163
  createImageInputStream, 572
  createImageOutputStream, 572
  createNewFile, 74
  createPrintJob, 613
  createStatement, 243; 268
  createSubcontext, 302
  createTextNode, 161; 163
  curveTo, 523; 532
  decode, 40; 225
  defaultPage, 598
  deleteRow, 269
  depthFirstEnumeration, 423; 427
  destroySubcontext, 302
  digest, 793
  doAs, 776
  doAsPrivileged, 776
  doFinal, 816
  doPost, 222
  draw, 517; 518; 533
  draw3DRect, 518
  drawArc, 518; 519
  drawLine, 518; 519
  drawOval, 518; 519
  drawPolygon, 518
  drawPolyline, 518
  drawRect, 516
  drawRectangle, 518; 519
  drawRoundRect, 518; 519
  encode, 225
  endDocument, 156
  endElement, 156
  equals, 331
  error, 127
  evaluate, 145; 149
  exceptionThrown, 734
  exclusiveOr, 533
  execute, 243; 272
  executeBatch(), 284
  executeQuery, 243; 259
  executeUpdate, 243; 259
  exportObject, 852
  fatalError, 127
  fill, 517; 518; 533
  fillOval, 516
  filter, 589
  findClass, 742; 748
  findColumn, 244
  fireIndexedPropertyChange, 691
  firePropertyChange, 691; 693
  fireVetoableChange, 692; 693
  first, 268
  flavorsChanged, 629
  flip, 88
  flush, 22
  format, 316
  generateSecret, 816
  getAdvance, 549
  getAll, 303
  getAllFrames, 512
  getAllowsChildren, 415
  getAllowsUserInteraction, 216

getAscent, 549
getAttribute, 119
getAttributes, 302; 621
getAuthority, 208
getAutoCommit, 284
getAvailableDataFlavors, 628
getAvailableIDs, 325
getAvailableLocales, 316; 324; 331
getBackground, 373
getBeanDescriptor, 708
getBeanInfo, 695
getBlockSize, 815
getBundle, 341
getCalendar, 324
getCategory, 621
getCellEditorValue, 406
getChild, 441
getChildAt, 422
getChildCount, 422; 441
getChildNodes, 110; 119
getClassLoader, 748
getClassName, 834
getClip, 549
getCodeSource, 757
getCollationKey, 331
getColumn, 393
getColumnClass, 391
getColumnCount, 282; 378
getColumnDisplaySize, 282
getColumnLabel, 282
getColumnName, 282
getColumnNumber, 127
getColumnType, 397
getCommand, 272
getComponentAt, 497
getConcurrency, 268
getConnection, 242
getConnectTimeout, 216
getContent, 217
getContentEncoding, 209; 212; 217
getContentLength, 209; 212; 217
getContentPane, 514
getContents, 623; 626
getContentType, 209; 212; 217
getContextClassLoader, 748
getCountry, 310
getCurrencyCode, 317
getCurrencyInstance, 316
getCustomEditor, 707
getData, 120; 627
getDataElements, 576; 580; 581
getDate, 209; 212; 217
getDateInstance, 324
getDateTimeInstance, 324
getDecompositon, 331
getDefault, 310
getDefaultEditor, 405
getDefaultFractionDigits, 318
getDefaultName, 785
getDefaultRenderer, 405
getDescent, 549
getDesktopPane, 514
getDisplayCountry, 310
getDisplayLanguage, 310
getDisplayName, 310; 325; 696
getDisplayVariant, 311
getDocumentElement, 110; 119
getDoInput, 216
getDoOutput, 216
getElementAt, 367
getErrorStream, 222; 225
getExceptionListener, 734
getExpiration, 209; 212; 217
getFilePointer, 44
getFileSuffixes, 573
getFirstChild, 119
getFontRenderContext, 549
getForeground, 373
getFormatNames, 573
getFragment, 208
getFrameIcon, 514
getHeaderField, 217
getHeaderFieldKey, 211; 217
getHeaderFields, 211; 217
getHeight, 573; 593; 599
getHost, 208
getIcon, 696
getIconAt, 497
getID, 303; 325
getIfModifiedSince, 216
getImageableHeight, 593
getImageableWidth, 593
getImageableX, 593; 599
getImageableY, 593; 599
getImageHeight, 599
getImageWidth, 599
getIndex, 692
getIndexedReadMethod, 697
getIndexedWriteMethod, 697
getIndexOfChild, 441
getInputStream, 185
getInstance, 557; 792
getJavaInitializationString, 707
getJDBCMajorVersion, 282
getJDBCMinorVersion, 282
getKeys, 341

getLanguage, 310
getLastChild, 119
getLastModifed, 217
getLastModified, 209; 212
getLastSelectedPathComponent, 422
getLayoutOrientation, 362
getLeading, 549
getLength, 120; 156
getLineNumber, 127
getListCellRendererComponent, 373
getLocale, 333; 354
getLocalName, 151; 156
getMaxConnections, 282
getMaximum, 487
getMaximumIntegerDigits, 316
getMaxStatements, 245; 282
getMetaData, 281; 282
getMimeType, 628
getMIMETypes, 573
getMinimum, 487
getMinimumFractionDigits, 316
getMinimumIntegerDigits, 316
getModel, 368
getName, 621; 696; 771; 777; 785; 834
getNamespaceURI, 151
getNewValue, 514; 692
getNextSibling, 119
getNodeName, 119; 151
getNodeValue, 119
getNumberFormat, 324
getNumberInstance, 316
getNumImages, 573
getNumThumbnails, 573
getObject, 341
getOldValue, 692
getOrientation, 599
getOriginatingProvider, 573
getOutputStream, 220
getPageCount, 600
getParent, 422
getParentNode, 119
getPassword, 272; 786
getPath, 208; 434
getPaths, 434
getPercentInstance, 316
getPersistenceDelegate, 734
getPixel, 575; 580
getPort, 208
getPreviousSibling, 119
getPrincipals, 776
getPrintable, 609
getPrinterJob, 598
getPrintService, 614
getPrompt, 785; 786
getPropertyChangeEvent, 693
getPropertyChangeListeners, 691
getPropertyDescriptors, 700
getPropertyName, 514; 692
getPropertyType, 697
getProtectionDomain, 757
getPrototypeCellValue, 366
getQName, 157
getQuery, 208
getReaderFormatNames, 572
getReaderMIMETypes, 572
getReadMethod, 697
getReadTimeout, 217
getRepresentationClass, 628
getRequestProperties, 217
getResultSet, 244
getRGB, 576; 580; 581
getRoot, 441
getRotateInstance, 545; 546
getRow, 268
getRowCount, 378
getRowHeight, 392
getRowMargin, 392
getSavepointId, 285
getSavepointName, 285
getScaleInstance, 545; 546
getScheme, 208
getSchemeSpecificPart, 208
getSelectedIndex, 497
getSelectedValue, 362
getSelectedValues, 362
getSelectionBackground, 373
getSelectionForeground, 373
getSelectionMode, 362
getSelectionPath, 422; 433
getSelectionPaths, 433
getShearInstance, 545; 546
getShortDescription, 696
getSize, 367
getString, 341; 487
getStringArray, 341
getSubject, 776
getSystem, 853
getSystemClassLoader, 748
getSystemClipboard, 626
getTabCount, 498
getTableCellEditorComponent, 398; 406
getTableCellRendererComponent, 405
getTableName, 272
getTables, 281
getTagName, 110; 119; 151
getTimeInstance, 324
getTimeZone, 324; 325

getTitleAt, 497
getTransferData, 627
getTransferDataFlavors, 629
getTranslateInstance, 545; 547
getType, 268
getUpdateCount, 244
getURI, 156
getURL, 272; 472; 476
getUseCaches, 216
getUserInfo, 208
getUsername, 272
getValue, 157; 487
getValueAt, 378
getVariant, 311
getVetoableChangeListeners, 693
getVisibleRowCount, 361
getWidth, 573; 593; 599
getWriteMethod, 697
getWriterFormatNames, 572
getWriterMIMETypes, 572
handle, 785
handleGetObject, 341
humanPresentableName, 628
hyperlinkUpdate, 472; 475
implies, 757; 771
inDaylightTime, 325
indexOfTab, 498
init, 815; 816
initialize, 734; 735; 786
insertNodeInto, 423
insertRow, 269
insertTab, 497
instantiate, 721; 734; 735
intersect, 533
isAfterLast, 269
isBeforeFirst, 269
isCanceled, 488
isCellEditable, 396; 406
isClosable, 513
isClosed, 513
isContinuousLayout, 492
isDataFlavorAvailable, 626
isDataFlavorSupported, 627
isEchoOn, 786
isExpert, 696
isFirst, 269
isGroupingUsed, 316
isHidden, 696
isIcon, 513
isIconifiable, 513
isIgnoringElementContentWhitespace, 127
isIndeterminate, 487
isLast, 269
isLeaf, 415; 441
isLenient, 324
isMaximum, 513
isMimeTypeEqual, 628
isNamespaceAware, 155
isOneTouchExpandable, 492
isPaintable, 707
isParseIntegerOnly, 316
isResizable, 513
isSelected, 513
isStringPainted, 487
isUnresolved, 202
isValidating, 127; 155
isVisible, 514
item, 120
last, 268
lineTo, 523; 532
list, 834
loadClass, 742
loadImage, 696
login, 776; 786
logout, 776; 786
lookup, 302; 833; 834
lookupPrintServices, 611; 613
lostOwnership, 627
makeShape, 524
makeVisible, 422
mark, 22
markSupported, 22
modifyAttributes, 302
moveTo, 532
moveToBack, 513
moveToCurrentRow, 269
moveToFront, 513
moveToInsertRow, 269
newDocument, 160; 163
newDocumentBuilder, 118
newInputStream, 202
newInstance, 118; 149; 163
newOutputStream, 202
newSAXParser, 155
newTransformer, 163; 179
newXpath, 149
nodeChanged, 423
nodesChanged, 423
open, 202
openConnection, 209; 216
openInputStream, 217
openOutputStream, 217
openStream, 216
pageDialog, 593; 599
paintValue, 707
parse, 119; 180; 316

pathFromAncestorEnumeration, 424
populate, 272
postOrderEnumeration, 427
postOrderTraversal, 424
preOrderEnumeration, 427
preOrderTraversal, 424
prepareStatement, 259; 268
print, 30; 591; 598
printDialog, 598
printf, 32
println, 30
propertyChange, 691
putClientProperty, 415
quadTo, 523; 532
read, 21; 196
readBoolean, 42
readFixedString, 44
readObject, 67; 734
readThumbnail, 573
rebind, 834
register, 852
registerGroup, 853
relative, 268
releaseSavepoint, 285
reload, 423
remaining, 88
removeCellEditorListener, 406
removeColumn, 387
removeElement, 368
removeNodeFromParent, 423
removePropertyChangeListener, 691; 693; 709
removeTab, 497
removeTreeModelListener, 441
removeVetoableChangeListener, 692; 693
reset, 22; 793
reshape, 514
resolveEntity, 127
rollback, 285
rotate, 544; 547
run, 777
scale, 544; 547
scrollPathToVisible, 422
setAllowsChildren, 415
setAllowUserInteraction, 209; 216
setAsksAllowsChildren, 415
setAttribute, 163
setAttributeNS, 163
setAutoCommit, 284
setAutoResizeMode, 383; 391
setBottomComponent, 492
setCalendar, 324
setCellEditor, 405
setCellRenderer, 373; 405
setClip, 517; 549
setClosable, 513
setClosed, 513
setClosedIcon, 427
setCommand, 272
setComponentAt, 497
setComposite, 517; 552; 557
setConnectTimeout, 216
setContentHandler, 180
setContentPane, 514
setContents, 623
setContextClassLoader, 748
setContinuousLayout, 492
setDataElements, 580
setDecomposition, 331
setDefault, 310
setDefaultAction, 652
setDisplayName, 696
setDoInput, 209; 216
setDoOutput, 209; 216; 220; 222
setDragEnabled, 644
setDragMode, 512
setEntityResolver, 126
setErrorHandler, 126
setExceptionListener, 734
setExpert, 696
setFixedCellHeight, 367
setFixedCellWidth, 367
setFrameIcon, 514
setGroupingUsed, 316
setHeaderRenderer, 405
setHeaderValue, 406
setHidden, 696
setIcon, 513
setIconAt, 497
setIconifiable, 513
setIfModifiedSince, 209; 216
setIgnoringElementContentWhitespace, 127
setIndeterminate, 487
setInput, 572
setLayoutOrientation, 362
setLeafIcon, 427
setLeftComponent, 492
setLenient, 324
setLocale, 333
setMaximizable, 513
setMaximum, 476; 487; 513
setMaximumFractionDigits, 316
setMaximumIntegerDigits, 316
setMaxWidth, 383; 393
setMinimum, 476; 487
setMinimumIntegerDigits, 316
setMinWidth, 383; 393

setName, 696; 785
setNamespaceAware, 155
setNote, 488
setNumberFormat, 324
setObject, 709
setOneTouchExpandable, 492
setOpenIcon, 427
setOutput, 574
setOutputProperty, 163
setPage, 471; 475
setPageable, 600
setPaint, 517; 542
setParseIntegerOnly, 316
setPassword, 272; 786
setPersistenceDelegate, 734
setPixel, 580
setPreferredWidth, 383; 393
setPrintable, 599
setProgress, 488
setPrototypeCellValue, 366
setReader, 870
setReadTimeout, 217
setRequestProperty, 209; 217
setResizable, 383; 393; 513
setRightComponent, 492
setRootVisible, 415
setRowHeight, 392
setRowMargin, 392
setSavepoint, 285
setSelected, 513
setSelectedIndex, 497
setSelectionMode, 362; 393
setShortDescription, 696
setShowsRootHandles, 415
setStrength, 331
setString, 487
setStringPainted, 487
setStroke, 540
setTableName, 272
setText, 471
setTimeZone, 324
setTitleAt, 497
setTopComponent, 492
setToRotate, 545
setToRotation, 547
setToScale, 545; 547
setToShear, 547
setToTranslation, 545; 547
setTransferHandler, 643
setTransform, 517; 545; 547
setURL, 272
setUseCaches, 209; 216
setUsername, 272
setUserObject, 415
setValidating, 127; 155
setValue, 487
setVisible, 514
setVisibleRowCount, 361
setWidth, 393
setWriter, 870
shear, 544; 547
shouldSelectCell, 406
show, 514
skip, 22
startDocument, 156
startElement, 156
stopCellEditing, 406
subtract, 532
supportsBatchUpdates, 285
supportsCustomEditor, 707
supportsResultSetType, 269
toString, 40; 311
transform, 547
translate, 544; 547
treeNodesChanged, 435; 441
treeNodesInserted, 435; 441
treeNodesRemoved, 435; 441
treeStructureChanged, 435; 441
unbind, 834
unread, 29
update, 793
updateRow, 269
useDaylightTime, 325
valueChanged, 362; 428; 433
valueForPathChanged, 441
vetoableChange, 505; 514; 692
warning, 127
write, 21; 22; 196; 574
writeChars, 41
writeDouble, 41
writeFixedString, 44
writeInt, 41
writeObject, 67; 733
writeStatement, 734
writeUTF, 41
запроса GET, 219
запроса POST, 219
обработки областей, 532
Множество Мандельброта, 577
Модель
дерева, 407
иерархических данных, 407
списка, 362
таблицы, 374; 377
Модификатор, 887
Монитор задачи, 480

Н

Набор
  полномочий, 754
  символов, 37
  строк, 270
Настройщик, 707

О

Область, 532
Обход узлов дерева
  в обратном порядке, 424
  в глубину, 424
  в ширину, 423
Объект
  -заглушка (stub), 827
  -ключ сравнения, 327
  отображения узлов дерева, 425
  отображения ячеек списка, 368
  путь к объекту, 416
  сериализация объектов, 55
Операция
  перетаскивания, 638
  свертки, 584
Отсечение, 547

П

Пакет
  java.nio, 37; 80
  пакетное обновление, 283
  ресурсов, 337
Панель
  рабочего стола, 488; 498
  с вкладками, 488; 492
  с границей раздела, 488
Перемещение, 543
Перетаскивание, 638
  контуров, 506
Песочница, 754
Печать
  многостраничная, 599
  полосовой принцип, 592
  потококая служба печати, 614
Пиксель, 550
  целевой, 550
Поворот, 543
Подписание
  JAR-файла, 804
  кода, 803
Политика безопасности, 754
Полномочия, 755
Порт, 182
Поток
  входных данных (input stream), 20; 185
  выходных данных (output stream), 20; 185
Права доступа, 755
Правила композиции Портера–Даффа, 550
Предварительный просмотр, 601
Преобразование
  XSLT, 171
  абсолютных адресов в относительные, 208
  координат, 543
Привилегии, 755
Принципал, 773
Программа ZipTest, 50
Прозрачность, 549
Пространство имен, 150
Протокол
  SMTP, 202
  SOAP, 826
Прямоугольник привязки, 542
Псевдоним (alias), 151
Пул соединений, 287
Пункт, 593

Р

Расцветка, 541
Регулярные выражения, 91
  синтаксис, 93
Редактор
  реестра, 955
  свойств, 679; 697
  ячейки, 418
    пользовательский, 398
Режим
  декомпозиции, 326
  развертки, 811
  свертки, 811
Ресурс, 337
Роль, 777

С

Свертка, 583
Свойство
  временное, 722
  простое, 684
  связанное, 685
  с ограничениями, 687
Сглаживание, 559
Сдвиг, 543
Сервер
  IBM Tivoli Directory Server, 289
  LDAP, 289
  Microsoft Active Directory, 289
  Novell eDirectory, 289

OpenLDAP, 289
Sun Java System Directory Server, 289
приложений, 231
Сервлет, 218
Сериализация, 634
Синтаксический анализ, 109
Служба
JAAS, 771
печати, 610; 611
потококая, 614
реестра самозагрузки, 831
Соединение
косое, 534
сетевое, 182
скругленное, 534
фасетное, 534
Сокет, 185
прерываемый, 196
Список, 356
модели списков, 362
Ссылка на объект, 638
Строка, 232
подстановки, 334
Стандарт
региональный, 306; 307
Стык, 534
Субъект, 773
Схема
XML, 128
дополнения, 811
Сценарий, 218
компиляция, 872
механизм сценариев, 866
язык сценариев, 866

Т

Таблица, 232
Тип
штриха, 533
документа, 610
линии, 533
Точка
конечная, 522
контрольная, 522
сохранения, 283
Транзакция, 282
автоматическая фиксация, 283

У

Узел
корневой, 407
пользовательский объект узла, 408
Унифицированное имя ресурса (URN), 207
Унифицированный идентификатор ресурса (URI), 207
Унифицированный указатель ресурса (URL), 207

Ф

Файл
блокирование файлов, 89
манифеста, 675
политики, 758
прав доступа, 755
свойств, 104; 338
с произвольным доступом, 43
указатель файла, 44
Фасетное соединение, 534
Фасетный угол
предельный, 534
Фигура отсечения (clipping shape), 547
Фильтр, 581
размытия, 583
Фильтрация изображений, 581
Фрейм
внутренний, 488; 499
каскадное расположение, 501
мозаичное расположение, 501

Ц

Цифровая подпись, 738; 787

Ш

Шифр
симметричный, 810
Шифрование, 810
правила шифрования
базовые, 795
отличительные, 795
с открытым ключом, 793; 818
Шрифт
контекст отображения шрифтов, 548
Штрих, 533

Э

Элемент
дочерний, 107
корневой, 107
Эллипс, 519

Я

Язык
IDL, 826
SGML, 105
SQL, 228; 231
WSDL, 826
XML, 105
сценариев, 866

JAVA™ 2
БИБЛИОТЕКА ПРОФЕССИОНАЛА
ТОМ 1. ОСНОВЫ
8-Е ИЗДАНИЕ

Кей Хорстманн
Гари Корнелл



Книга ведущих специалистов по программированию на языке Java представляет собой обновленное издание фундаментального труда, учитывающее всю специфику новой версии платформы Java SE 6. Подробно рассматриваются такие темы, как организация и настройка среды программирования на Java, фундаментальные структуры данных, объектно-ориентированное программирование и его реализация в Java, интерфейсы, программирование графики, обработка событий, Swing, развертывание приложений и аплетов, отладка, обобщенное программирование, коллекции и построение многопоточных приложений. Книга изобилует множеством примеров, которые не только иллюстрируют концепции, но также демонстрируют способы правильной разработки, применяемые в реальных условиях. Книга рассчитана на программистов разной квалификации, а также будет полезна студентам и преподавателям.

www.williamspublishing.com

ISBN 978-5-8459-1378-4

в продаже

REKLAMA_CoreJava-v2.indd 984 09.10.2008 15:11:47

ПОЛНЫЙ СПРАВОЧНИК ПО JAVA™ 7-Е ИЗДАНИЕ

Герберт Шилдт



www.williamspublishing.com

Книга известного “гуру” в области программирования посвящена новой версии одного из наиболее популярных и совершенных языков – Java. Построенная в виде учебного и справочного пособия, она является превосходным источником исчерпывающей информации по последней версии платформы Java, Java SE 6, и позволяет практически с нуля научиться разрабатывать приложения и аплеты производственного качества. Помимо синтаксиса самого языка и фундаментальных принципов программирования, в книге подробно рассматриваются такие сложные вопросы, как ключевые библиотеки Java API, каркас коллекций, создание аплетов и сервлетов, AWT, Swing и Java Beans. Немалое внимание уделяется вводу-выводу, работе в сети, регулярным выражениям и обработке строк.

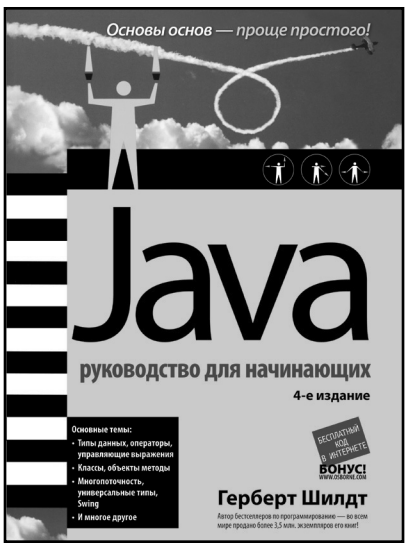
ISBN 978-5-8459-1168-1

в продаже

REKLAMA_CoreJava-v2.indd 985 09.10.2008 15:11:48

JAVA™
РУКОВОДСТВО ДЛЯ НАЧИНАЮЩИХ

Герберт Шилдт



www.williamspublishing.com

Книга разделена на 15 модулей. В начале автор знакомит читателя с основными элементами языка: типами данных, операторами, управляющими выражениями. Здесь же приведена информация об объектных средствах: классах, объектах и методах. Далее следуют вопросы, не владея которыми невозможно написать даже простые программы: доступ к объектам и членам класса, особенности использования ссылок, средства ввода-вывода, поддержка событий, обработка исключений и т.д. Большое внимание уделяется объектно-ориентированному программированию, рассматриваются инкапсуляция, наследование и полиморфизм. Несмотря на то что большая часть книги посвящена Java-приложениям, читатель также найдет в ней сведения об аплетах. Помимо базовых вопросов Java-программирования, в книге рассматриваются элементы, реализованные в последних версиях Java, например универсальные классы и нумерованные типы. Завершается книга рассмотрением Swing — мощного инструмента создания графических пользовательских интерфейсов.

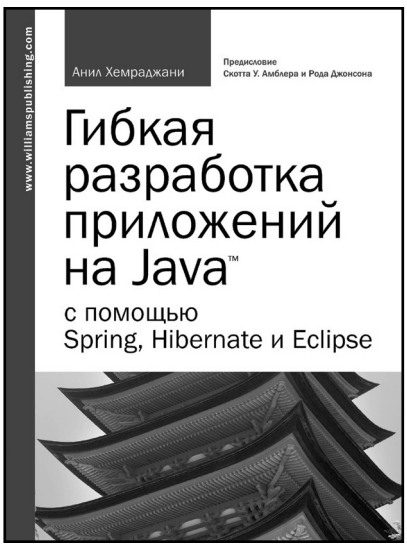
ISBN 978-5-8459-1440-8

в продаже

REKLAMA_CoreJava-v2.indd 986 09.10.2008 15:11:49

ГИБКАЯ РАЗРАБОТКА ПРИЛОЖЕНИЙ НА JAVA
С ПОМОЩЬЮ SPRING, HIBERNATE И ECLIPSE

Анил Хемраджани



www.williamspublishing.com

В этой книге основное внимание уделено разработке и в меньшей степени инфраструктуре. Другими словами, больше внимания уделено технологиям разработки приложений, таким как Spring, Hibernate и Eclipse, а не программным продуктам, таким как серверы приложений или базы данных. Все, что представлено в этой книге, опробовано в реальных приложениях, которые успешно работают (некоторые в кластеризуемой среде сервера приложений). Одна из задач этой книги заключается в краткости и конкретности, поэтому автор решил практически полностью сосредоточиться на разработке хорошо масштабируемого приложения. В данной книге, кроме технологий Spring, Hibernate и Eclipse, также описаны альтернативные и конкурирующие технологии.

ISBN 978-5-8459-1375-3 **в продаже**

REKLAMA_CoreJava-v2.indd 987 09.10.2008 15:11:50

JAVA МЕТОДИКИ ПРОГРАММИРОВАНИЯ ШИЛДТА

Герберт Шилдт



www.williamspublishing.com

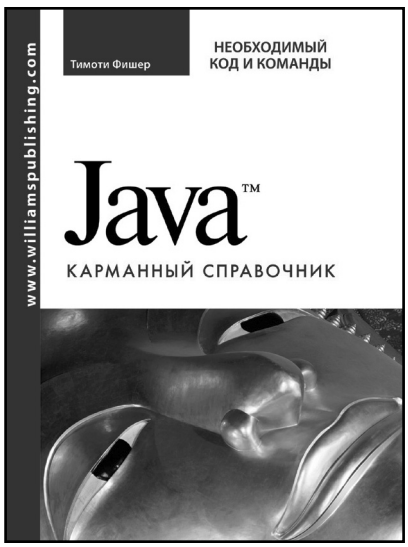
Книга общепризнанного авторитета в области языков программирования построена в виде готовых рецептов, иллюстрирующих передовые приемы и подходы в программировании на Java. В каждом рецепте описан набор ключевых ингредиентов, таких как классы, интерфейсы и методы. Затем даются шаги, которые потребуется предпринять, собирая из этих ингредиентов последовательность кода, необходимого для достижения нужного результата. Подробно рассматриваются такие темы, как обработка строк, использование регулярных выражений, работа с файлами, создание аплетов и сервлетов, построение графических интерфейсов с помощью Swing, применение каркаса коллекций, многопоточность и форматирование данных. Книга рассчитана на программистов разной квалификации, а также будет полезна студентам и преподавателям дисциплин, связанных с программированием на языке Java.

ISBN 978-5-8459-1395-1 **в продаже**

REKLAMA_CoreJava-v2.indd 988 09.10.2008 15:11:51

JAVA
КАРМАННЫЙ СПРАВОЧНИК

Тимоти Фишер



www.williamspublishing.com

В данную книгу включено более 100 фрагментов кода, с помощью которых читатель сможет быстро реализовать требуемые ему функциональные возможности на языке Java: обработку текстовых строк, чисел и дат; поиск по образцу с помощью регулярных выражений; работу с базами данных и XML; создание многопоточных приложений; выполнение операций ввода-вывода; создание клиентских и серверных сетевых приложений и др. Приведенные в ней примеры упорядочены по принципу "от простого к сложному" и отражают практически весь спектр задач, с которыми обычно сталкивается программист при создании типичного приложения на Java. Настоящая книга — не учебник по языку Java, не введение в него и, тем более, не полное руководство. В книге описано лишь ограниченное количество классов и API. Эта книга будет полезна как для опытных программистов на Java, так и для новичков.

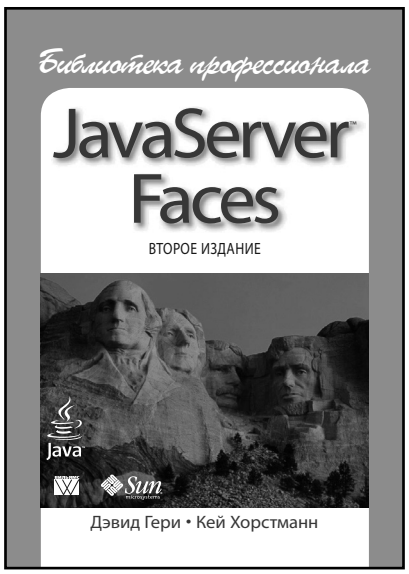
ISBN 978-5-8459-1392-0

в продаже

REKLAMA_CoreJava-v2.indd 989 09.10.2008 15:11:52

JAVASERVER FACES™ БИБЛИОТЕКА ПРОФЕССИОНАЛА
2-Е ИЗДАНИЕ

Дэвид Гери
Кей Хорстманн



www.williamspublishing.com

Эта книга представляет собой руководство для Web-разработчиков, занимающихся дизайном пользовательских интерфейсов, а также для программистов, создающих компоненты многократного пользования для Web-приложений. Подробно рассматриваются такие темы, как подготовка среды программирования, связывание JSF-дескрипторов с логикой приложения, страничная навигация, использование стандартных дескрипторов JSF, преобразование и проверка достоверности входных данных. Немалое внимание уделено способам уменьшения сложности приложений серверной стороны, обработке событий, реализации специальных компонентов, подключению к базам данных и каталогам LDAP, использованию Ajax и применению технологий с открытым исходным кодом, среди которых Facelets, Seam и Shale. Книга рассчитана на программистов разной квалификации, а также будет полезна студентам и преподавателям

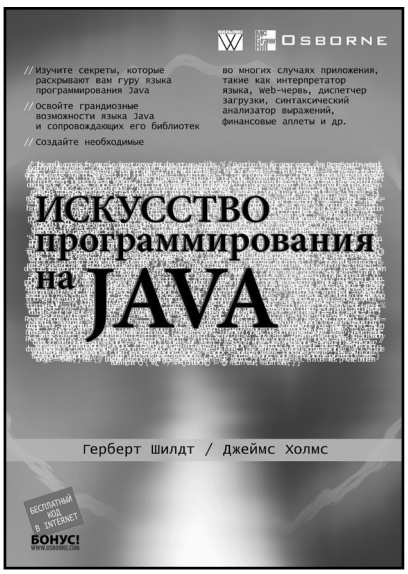
ISBN 978-5-8459-1396-8

в продаже

REKLAMA_CoreJava-v2.indd 990 09.10.2008 15:11:53

ИСКУССТВО ПРОГРАММИРОВАНИЯ НА JAVA

Герберт Шилдт
Джеймс Холмс



www.williamspublishing.com

Эта книга отличается от множества других книг по языку Java. В то время как другие книги обучают основам языка, эта книга показывает, как использовать язык наиболее эффективно, с большей пользой и отдачей для решения запутанных задач программирования. На страницах книги постепенно раскрывается мощь, универсальность и элегантность языка Java.
Как и можно ожидать, несколько описанных приложений связаны непосредственно с Internet. Многие главы посвящены анализу кода. В них показаны потрясающие возможности языка Java при создании приложений для Internet. Легкость, с которой эти программы могут быть написаны на языке Java, подтверждает гибкость и элегантность языка.
В каждой главе рассматриваются фрагменты кода, которые без изменений вы сможете использовать в своих программах. Например, синтаксический анализатор может послужить отличным дополнением для многих разработок.
Книга рассчитана на студентов, преподавателей и специалистов в области компьютерных технологий.

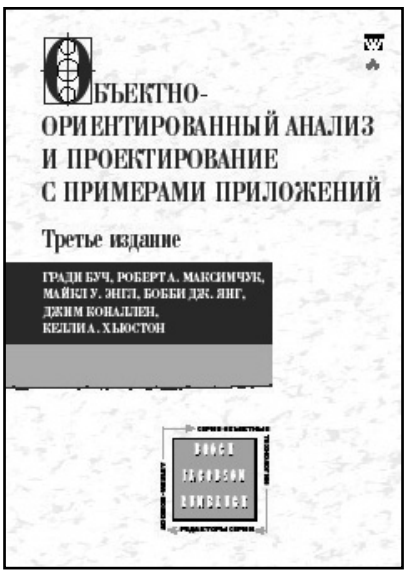
ISBN 5-8459-0786-1

в продаже

REKLAMA_CoreJava-v2.indd 991 09.10.2008 15:11:53

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ С ПРИМЕРАМИ ПРИЛОЖЕНИЙ
ТРЕТЬЕ ИЗДАНИЕ

Гради Буч,
Роберт А. Максимчук,
Майкл У. Энгл
и др.



www.williamspublishing.com

Книга представляет собой новое издание бестселлера Гради Буча по объектно-ориентированному анализу и проектированию. Авторы описывают объектные методы решения сложных проблем, связанные с разработкой систем и программного обеспечения. Используя многочисленные примеры, они иллюстрируют основные концепции объектно-ориентированного подхода на примере разработки систем управления, сбора данных и искусственного интеллекта. Читатели найдут в книге практические советы, касающиеся важных вопросов анализа, проектирования, реализации и оптимального управления проектами. Книга будет полезна системным аналитикам и архитекторам, программистам, преподавателям и студентам высших учебных заведений, а также всем специалистам по информационным технологиям.

ISBN 978-5-8459-1401-9 **в продаже**

REKLAMA_CoreJava-v2.indd 992 09.10.2008 15:11:54