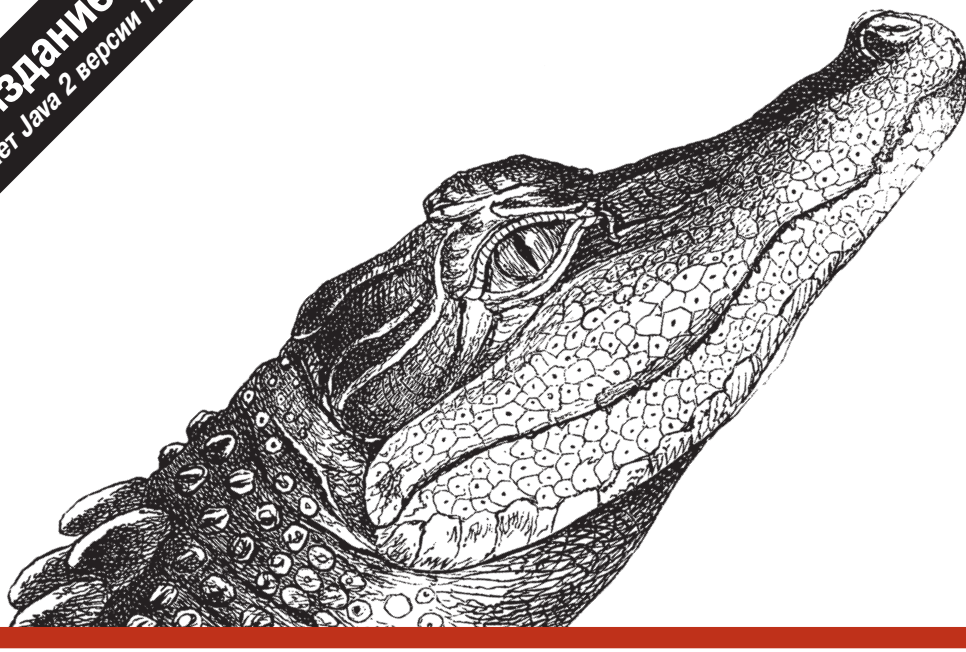
2-е издание

Охватывает Java 2 версии 1.3



JAVA™ В ПРИМЕРАХ

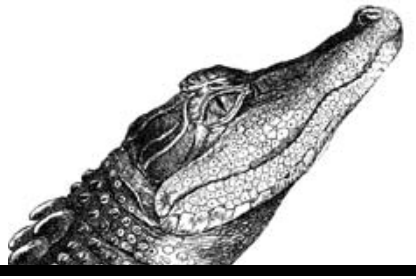
СПРАВОЧНИК

Учебное пособие к книге «Java. Справочник»

СИМВОЛ®

O'REILLY®

Дэвид Флэнаган



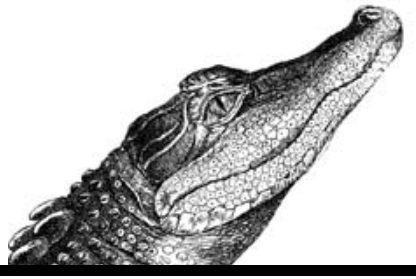
JAVA EXAMPLES IN A NUTSHELL

A Tutorial Companion to Java in a Nutshell

Second Edition

David Flanagan

O'REILLY®



JAVA В ПРИМЕРАХ

СПРАВОЧНИК

Учебное пособие к книге «Java. Справочник»

Второе издание

Дэвид Флэнаган

Санкт-Петербург — Москва
2003

Дэвид Флэнаган

Java в примерах. Справочник, 2-е издание

Перевод И. Асеева и И. Васильева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научные редакторы	*И. Васильев* *В. Шальнев*
Редактор	*В. Кузнецов*
Корректор	*С. Беляева*
Верстка	*Н. Гриценко*

Флэнаган Д.

Java в примерах. Справочник, 2-е издание – Пер. с англ. – СПб: Символ-Плюс, 2003. – 664 с., ил.

ISBN 5-93286-042-1

Второе издание книги «Java в примерах. Справочник» содержит 164 законченных практических примера: свыше 17 900 строк тщательно прокомментированного, профессионально написанного Java-кода, работающего с 20 различными программными интерфейсами Java, такими как сервлеты, JSP, XML, Swing и Java 2D. Приведены примеры, иллюстрирующие ключевые интерфейсы Java для корпоративных проектов, включая вызов удаленных методов (RMI), доступ к базам данных (JDBC). Автор бестселлера «Java in a Nutshell» (в русском переводе «Java. Справочник», Символ-Плюс) создал целую книгу примеров программ, на которых можно учиться и которые можно модифицировать для использования в своих приложениях. Если вы предпочитаете учиться «на примерах», эта книга для вас.

Книга дополняет серию справочников по Java издательства O'Reilly и будет полезна как начинающим, так и опытным Java-программистам. Удобный указатель примеров (глава 20) позволяет быстро найти метод или класс Java, а затем отыскать примеры, демонстрирующие их применение.

ISBN 5-93286-042-1
ISBN 0-596-00039-1 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 13.10.2003. Формат 70x100 1/16 . Печать офсетная.
Объем 41,5 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с диапозитивов в Академической типографии «Наука» РАН
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие

Эта книга из той же серии, что и мои предыдущие книги «Java in a Nutshell»[1], «Java Foundation Classes in a Nutshell» и «Java Enterprise in a Nutshell». Хотя эти книги являются, по существу, справочниками, они также содержат краткие введения в различные темы, относящиеся к программированию на Java™, и небольшие наборы примеров программ. Я писал книгу «Java в примерах», чтобы продолжить начатые в тех книгах темы, предоставив подборку примеров программ, полезных как для начинающих, так и для опытных программистов на Java.

Писать эту книгу было очень забавно. Первое издание по времени почти совпало с выходом Java версии 1.1, которая по размеру превосходила версию Java 1.0 более чем в 2 раза. Пока я был занят написанием дополнительных примеров для второго издания «Java in a Nutshell», разработчики корпорации Sun были заняты превращением Java в нечто такое, что больше не могло уложиться в рамки справочника. Из-за этого раздел, содержащий краткий справочник, так разросся, что книга «Java in a Nutshell» уже не могла вместить много примеров. Мы могли бы включить некоторые примеры по новым возможностям Java 1.1, но нам пришлось бы урезать гораздо больше, чем мы могли вставить. Это было трудное решение; примеры в «Java in a Nutshell» были одной из самых популярных ее составляющих.

Эта книга – результат тех сокращений, и я весьма доволен решением, которое мы тогда приняли. Получив свободу посвятить примерам всю книгу, я смог написать те примеры, которые я действительно хотел написать. Я смог погрузиться в тему так глубоко, как никогда раньше, и я по-настоящему наслаждался исследованием и экспериментами,

[1] Дэвид Флэнаган «Java. Справочник», 4-е издание, Символ-Плюс, 2003.

которые проводил при создании примеров. Для второго издания книги я имел удовольствие исследовать и экспериментировать с новыми разделами Java API: Swing™, Java 2D™, сервлетами и XML. Я надеюсь, что вы будете использовать эти примеры как отправные точки для ваших собственных исследований и ощутите вкус такого же волнения, какое я испытывал при их написании.

Как и сказано в ее названии, обучение в этой книге строится на примерах, на которых большинство людей обучается быстрее всего. Она не ведет вас за руку и не содержит подробного описания точного синтаксиса и операторов Java. Эта книга предназначена для совместной работы с книгами «Java in a Nutshell», «Java Foundation Classes in a Nutshell» и «Java Enterprise in a Nutshell». Изучая примеры из нее, вы наверняка оцените полезность этих изданий. Вас могут также заинтересовать и другие книги издательства O'Reilly из серий, посвященных языку Java. Список этих книг находится на сайте *http://java.oreilly.com*.

Эта книга состоит из трех частей. В главах с 1 по 9 рассматриваются основные неграфические разделы Java API. API, упомянутые в этих главах, документированы в книге «Java in a Nutshell». Главы с 10 по 15, составляющие вторую часть книги, демонстрируют графику Java и API графического пользовательского интерфейса, которые подробно рассмотрены в «Java Foundation Classes in a Nutshell». Наконец, главы с 16 по 19 содержат примеры Java API для корпоративных проектов и дополняют книгу «Java Enterprise in a Nutshell».

Вы можете читать главы этой книги в более или менее произвольном порядке, в каком они заинтересуют вас. Однако между главами имеются некоторые взаимозависимости, и некоторые из глав действительно нужно читать в том порядке, в котором они представлены. Например, прежде чем переходить к главе 5 «Сетевые операции», важно изучить главу 3 «Ввод/вывод». Глава 1 «Основы Java» и глава 2 «Объекты, классы и интерфейсы» предназначены для программистов, только начинающих работать с Java. Опытные Java-программисты, скорее всего, захотят пропустить их.

Примеры Java online

Примеры этой книги доступны в Интернете, поэтому вам не нужно набирать их все вручную! Вы можете загрузить их с веб-сайта автора *http://www.davidflanagan.com/javaexamples2* или с сайта издателя *http://www.oreilly.com/catalog/jenut2*.[1] На сайте издателя вы также можете найти список обнаруженных ошибок и опечаток. Примеры являются бесплатными для некоммерческого использования. Однако при желании использовать их в коммерческих целях необходимо оп-

[1] Речь здесь и ниже идет об оригинальном издании от O'Reilly. – *Примеч. ред.*

латить номинальную стоимость коммерческой лицензии. За подробной информацией по лицензированию обращайтесь на сайт *http://www.davidflanagan.com/javaexamples2*.

Другие книги от O'Reilly

Издательство O'Reilly издает все серии книг по Java. В их число входят «Java in a Nutshell», «Java Foundation Classes in a Nutshell» и «Java Enterprise in a Nutshell», которые, как сказано выше, являются спутниками этой книги.

С данной книгой связан справочник «Java Power Reference». Это электронный справочник по Java на компакт-диске, выполненный в стиле «Java in a Nutshell». Но будучи разработанным для просмотра в броузере, он полностью оснащен гиперссылками и включает мощный механизм поиска. Он шире по охвату, но менее глубок, чем книги серии «Java in a Nutshell». Справочник «Java Power Reference» охватывает все API платформы Java 2 и, кроме того, API многих стандартных расширений. Но он не предоставляет учебных разделов по различным API и не содержит описаний отдельных классов.

Вы можете найти полный список книг по Java от издательства O'Reilly на сайте *http://java.oreilly.com*. Отдельные главы этой книги ссылаются на специализированные книги, которые помогут вам изучить этот материал более подробно.

Соглашения, используемые в этой книге

В этой книге используются следующие соглашения по шрифтовому оформлению:

Курсивное начертание

Применяется для выделения и указания первого вхождения термина. Курсивом также выделяются команды, адреса электронной почты, веб-сайты, FTP-сайты, имена файлов и каталогов, группы новостей.

Полужирное начертание

Иногда используется для выделения клавиш клавиатуры или элементов пользовательского интерфейса, таких как кнопка **Back** или меню **Options**.

`Моноширинный шрифт`

Используется во всем Java-коде и вообще для всего, что вы набираете на клавиатуре при программировании, включая ключевые слова, типы данных, константы, имена методов, переменные, имена классов и интерфейсов. Кроме того, используется для командных строк и параметров, которые должны печататься на экране дословно, а также для тегов, которые могут появляться в HTML-документе.

Наклонный моноширинный шрифт

Используется для выделения имен параметров методов, а во многих случаях – в качестве метки-заполнителя, указывающей элемент, который в вашей программе должен быть заменен фактическим значением. Также используется для переменных выражений в параметрах командной строки.

Присылайте комментарии

Информация в этой книге была просмотрена и проверена, но вы можете обнаружить, что некоторые средства изменились (или даже найти ошибки!). Вы можете послать сообщение о любых найденных вами ошибках, а также предложения по будущим изданиям по адресу:

O'Reilly & Associates, Inc.
101 Morris Street
Sebastopol, CA 95472
(800) 998-9938 (для США и Канады)
(707) 829-0515 (международный/местный)
(707) 829-0104 (факс)

Вы можете посылать и электронные сообщения. Для внесения вас в список рассылки или запроса каталога отправьте электронное письмо по адресу:

info@oreilly.com

Чтобы задать технический вопрос или отправить комментарий по данной книге, отправьте электронное письмо по адресу:

bookquestions@oreilly.com

На веб-сайте этой книги находятся список примеров, опечаток и планы на будущие издания. Прежде чем отправить сообщение об ошибке, проверьте список опечаток, чтобы выяснить, нет ли там уже сообщения о ней. К этой странице вы можете обратиться по адресу:

http://www.oreilly.com/catalog/jenut2

Дополнительную информацию по этой и другим книгам можно получить на веб-сайте компании O'Reilly:

http://www.oreilly.com

Благодарности

Мои благодарности, как всегда, редактору Пауле Фергюсон (Paula Ferguson), собравшей все в одну последовательную книгу и терпимой к моим многократным нарушениям графика. Благодарю также Франка Виллисона (Frank Willison) и Тима О'Рейли (Tim O'Reilly) за их желание и энтузиазм взяться за книгу, состоящую только из примеров.

В работе над этой книгой мне помогали авторы других книг по Java из издательства O'Reilly. Джонатан Нудсен (Jonathan Knudsen), автор нескольких книг по Java от O'Reilly, просмотрел главы по графике и печати. Боб Экштейн (Bob Eckstein), соавтор книги «Java Swing», просмотрел главу по Swing. Джейсон Хантер (Jason Hunter), автор книги «Java Servlet Programming», просмотрел главу по сервлетам. Ханс Бергстен (Hans Bergsten), автор выходящей книги по JavaServer Pages™, также просмотрел главу по сервлетам, но особенно тщательно изучил примеры по JSP. Бретт Мак-Лахлин (Brett McLaughlin), автор книги «Java and XML»[1], просмотрел главу по XML. Джордж Риз (George Reese), автор книги «Database programming with JDBC and Java», любезно согласился просмотреть главу по базам данных. Джим Фарли (Jim Farley), автор книги «Java Distributed Computing» и соавтор книги «Java Enterprise in a Nutshell», просмотрел примеры по RMI. Мастерство, приложенное этими рецензентами, значительно улучшило качество моих примеров. Я обязан им всем и настоятельно рекомендую их книги!

Группа подготовки издания в O'Reilly&Associates в очередной раз выполнила грандиозную работу по превращению предоставленной мною рукописи в превосходную книгу. И, как обычно, я благодарен и восхищаюсь ими.

В заключение хочу поблагодарить Кристи (Christie) – по причинам, которых слишком много, чтобы приводить их здесь.

Дэвид Флэнаган (David Flanagan)
http://www.davidflanagan.com
Июль 2000

[1] Брет Мак-Лахлин «Java и XML», 2-е издание, Символ-Плюс, 2002.

I

Основные Java API

Часть I содержит примеры, демонстрирующие основные возможности Java и базовые программные интерфейсы Java. Эти примеры соответствуют той части языка, которая рассматривается в книге «Java in a Nutshell».

Глава 1

Основы Java

В этой главе содержатся примеры, демонстрирующие основы синтаксиса языка Java; подразумевается, что она изучается вместе с главой 2 книги «Java in a Nutshell»[1]. Если у вас есть существенный опыт программирования на C или C++, вы легко поймете материал этой главы. Если же вы переходите на Java с другого языка, вам, возможно, понадобится изучить эти примеры более тщательно.

Важнейшим шагом при изучении нового языка программирования является овладение основными управляющими операторами языка. В отношении Java это означает изучение оператора ветвления `if/else` и операторов цикла `while` и `for`. Учиться программировать хорошо – это как учиться решать словесные задачи на уроках алгебры в средней школе: вы должны перевести абстрактное описание задачи на конкретный математический язык (в нашем случае на язык Java). Когда вы научитесь мыслить категориями операторов `if`, `while` и `for`, другие операторы Java, такие как `break`, `continue`, `switch` и `try/catch/finally`, будет уже легко усвоить. Обратите внимание на то, что хотя Java – это объектно-ориентированный язык, мы не будем обсуждать объекты до главы 2.

Итак, приняв изложенное за введение и положив себе целью изучить основы синтаксиса, перейдем прямо к написанию программ на Java.

Hello World

Еще в 1978 году Брайан Керниган (Brian Kernighan) и Деннис Ричи (Dennis Ritchie) в своей классической книге «Язык программирования C»

[1] Дэвид Флэнаган «Java. Справочник», 4-е издание, Символ-Плюс, 2003

писали: «Программа, которую следует написать первой, для всех языков одна». Разумеется, они имели в виду программу «Hello World». Реализация программы «Hello World» на языке Java показана в примере 1.1.

Пример 1.1. Hello.java

```
package com.davidflanagan.examples.basics;     // Уникальный префикс имени
                                               // класса
public class Hello {                           // В Java все является классом
    public static void main(String[] args) {   // Все программы должны
                                               // содержать метод main()
    System.out.println("Hello World!");        // Говорим Hello!
    }                                          // Отмечает конец main()
}                                              // Отмечает конец класса
```

Первой строкой программы является объявление `package`. Оно указывает имя *пакета* (*package*), в который входит эта программа. Именем программы (как мы это видим из второй строки) является `Hello`. Имя пакета – это `com.davidflanagan.examples.basics`. Мы можем соединить их, чтобы получить полное имя программы, `com.davidflanagan.examples.basics.Hello`. Применение пакетов обеспечивает каждую программу Java уникальным пространством имен. Поместив программу `Hello` в пакет, я гарантирую, что если еще кто-нибудь назовет свою программу `Hello`, конфликта имен не произойдет. Чтобы обеспечить заведомую уникальность имени пакета, я следую соглашению, по которому имя пакета начинается с доменного имени в Интернете, записанного в обратном порядке.[1] В каждой главе этой книги используется свой пакет с именем, начинающимся с префикса `com.davidflanagan.examples`. В этом примере, поскольку глава посвящена основам языка Java, именем пакета будет `com.davidflanagan.examples.basics`.

Ценность программы «Hello World» заключается в том, что она является шаблоном, который вы можете расширять в собственных экспериментах с Java. Вторая и третья строки в примере 1.1 являются обязательной составляющей шаблона. Любая программа – на самом деле, любой кусок кода на Java – является классом. Во второй строке программы говорится, что мы создаем класс `Hello`. Она также сообщает, что наш класс является внешним (`public`); это означает, что его может использовать кто угодно.

Всякая самостоятельная программа на Java обязательно содержит метод `main()`. Именно с этого места интерпретатор Java начинает исполнение программы, написанной на Java. В третьей строке примера объявляется метод `main()`. В ней говорится, что этот метод является открытым (`public`), что он не возвращает значения (или, другими словами,

[1] Мой веб-сайт называется *http://www.davidflanagan.com*, поэтому имя моего пакета начинается с `com.davidflanagan`.

возвращает значение void) и что в качестве аргументов он получает строковый массив. Имя этого массива – args. В этой строке утверждается также, что метод main() является статическим (static). (В этой главе мы будем иметь дело исключительно со статическими методами. В главе 2, когда мы начнем работать с объектами, вы узнаете, что такое статические методы и что методы, как правило, являются, напротив, нестатическими.)

Вы, во всяком случае, можете попросту заучить следующую строку:

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

Всякая самостоятельная программа на Java содержит строку, которая выглядит в точности, как эта. (По правде говоря, строковый массив вы можете назвать, как вам заблагорассудится, но обычно его называют именно args.)

Пятая и шестая строки примера 1.1 просто отмечают конец метода main() и класса Hello. Подобно большинству современных языков, Java является языком с блочной структурой. Это означает, что у таких элементов, как классы и методы, есть тело, являющееся «блоком» программного кода. В Java начало блока отмечается открывающей фигурной скобкой {, а конец – закрывающей }. Блоки методов всегда определяются внутри блоков классов и, как мы увидим из нижеследующих примеров, блоки методов могут содержать операторы if и циклы for, образующие внутри метода подблоки. Более того, такие блоки операторов могут вкладываться друг в друга на любую глубину.

Три первые и две последние строки примера 1.1 принадлежат каркасу приложения на Java. Интерес представляет только четвертая строка примера. Эта строка печатает слова «Hello World!». Метод System.out.println() отправляет выводимую строку в «стандартный вывод», которым обычно является экран. Этот метод применяется на протяжении этой и многих последующих глав этой книги. Но только в главе 3 «Ввод/вывод» вы поймете, что он делает в действительности. Если вам не терпится это узнать, найдите классы java.lang.System и java.io.PrintStream в книге «Java in a Nutshell» (или в каком-нибудь другом справочном руководстве по Java).

Последнее, о чем следует упомянуть в связи с этой программой, – это комментарии. В примере 1.1 применяются комментарии в стиле C++, которые начинаются с символов // и продолжаются до конца строки. Таким образом, все, что находится между символами // и концом строки, игнорируется компилятором Java. Вы обнаружите, что примеры в этой книге тщательно откомментированы. Комментарии в кодах заслуживают внимания, поскольку затрагивают некоторые вопросы, не нашедшие отражения в тексте.

Запуск программы «Hello World»

Чтобы запустить программу, нужно первым делом ее набрать.[1] Наберите при помощи текстового редактора программу `Hello` в том виде, в котором она приведена в примере 1.1. Но опустите теперь объявление пакета в первой строке. Сохраните программу в файле *Hello.java*.

Второй шаг состоит в компиляции программы. Если вы используете Java Software Development Kit (SDK) производства компании Sun, то будете компилировать программу командой *javac*.[2] Перейдите в каталог, в котором содержится файл *Hello.java*, и наберите эту команду (в предположении, что путь к *javac* известен системе):

```
% javac Hello.java
```

Если пакет Java SDK был правильно установлен, *javac* через небольшой промежуток времени создаст файл с именем *Hello.class*. В этом файле будет содержаться откомпилированная версия программы. Я уже говорил, что все, что вы пишете в Java, является классом, и этот факт отражен в расширении *.class* имени вашего файла. Важное правило, связанное с компиляцией программ на Java, состоит в том, что имя файла за вычетом расширения *.java* должно совпадать с именем определенного в файле класса. Таким образом, если бы вы набрали пример 1.1 и сохранили его в файле с названием *HelloWorld.java*, вам не удалось бы его откомпилировать.

Чтобы запустить программу (с применением Java SDK), наберите:

```
% java Hello
```

Эта команда произведет следующий вывод:

```
Hello World!
```

Команда *java* – это интерпретатор Java; он запускает виртуальную машину Java. Вы передаете *java* имя запускаемого класса. Обратите внимание на то, что при этом следует указывать имя класса `Hello`, а не имя файла *Hello.class*, содержащего скомпилированный класс.

Выше было показано, как компилируется и запускается программа на Java, не содержащая объявления пакета. Если вы опустили объявление `package` при наборе *Hello.java*, этих инструкций достаточно (если это не так, проверьте, правильно ли вы набрали программу). На прак-

---

[1] Хотя этот пример включен в онлайновый архив примеров, я советую вам набрать его самостоятельно. Если вы это сделаете, выражения языка Java глубже запечатлятся в вашей памяти. Я также буду предлагать вам модифицировать этот пример при объяснении некоторых аспектов исполнения программы.

[2] Если вы применяете другую среду разработки программ на Java, изучите инструкции производителя по компиляции и запуску программ и следуйте им.

тике, однако, все нетривиальные программы на Java (включая примеры из этой книги) содержат объявления `package`. При наличии объявления пакета процесс компиляции и запуска программ на Java несколько усложняется. Как я уже отмечал, программа на Java должна записываться в файл с именем, совпадающим с именем класса. Когда класс включается в пакет, добавляется требование, состоящее в том, чтобы имя каталога, в котором хранится этот файл, совпадало с именем пакета.

Давайте вернем в *Hello.java* объявление пакета:

```
package com.davidflanagan.examples.basics;
```

Создайте теперь каталог (или папку), в котором вы будете хранить все примеры из этой книги. В системе Windows, например, вы можете создать папку *c:\jenut2*. Под Linux можно создать каталог *~/jenut2*. В этом каталоге создайте подкаталог *com*. Теперь в каталоге *com* создайте подкаталог с именем *davidflanagan*, а затем создайте подкаталог *examples* в *davidflanagan*. Наконец, создайте в *examples* подкаталог *basics*. Скопируйте программу *Hello.java* (содержащую объявление пакета) в этот последний каталог. Под Windows получится файл с именем:

```
c:\jenut2\com\davidflanagan\examples\basics\Hello.java
```

Создав структуру каталогов и разместив в ней программу на Java, вы должны будете сообщить компилятору и интерпретатору Java, где теперь ее искать. Компилятору и интерпретатору нужно знать только выбранный вами базовый каталог; файл *Hello.class* они будут искать в его подкаталоге, ориентируясь по имени пакета. Чтобы указать Java направление поиска, вы должны присвоить значение переменной окружения `CLASSPATH` способом, соответствующим вашей операционной системе. Если, работая под Windows, вы приняли предложенное мной имя папки (*c:\jenut2*), можно использовать следующую команду:

```
C:\> set CLASSPATH=.;c:\jenut2
```

Эта команда предлагает Java искать программу сперва в текущем каталоге (`.`), а затем в каталоге *c:\jenut2*.

Под Unix при использовании оболочки *csh* можно написать следующую команду:

```
% setenv CLASSPATH .:/home/david/jenut2
```

В оболочках *sh* или *bash* соответствующая команда будет такой:

```
$ CLASSPATH=.:/home/david/jenut2; export CLASSPATH
```

Можно автоматизировать этот процесс, включив команду установки переменной `CLASSPATH` в стартовый файл, в *autoexec.bat* под Windows или в *.cshrc* под Unix (с оболочкой *csh*). Установив значение `CLASSPATH`, вы можете двинуться дальше и откомпилировать и запустить програм-

му `Hello`. Перед компиляцией перейдите в каталог *examples/basics*, содержащий *Hello.java*. Запустите компилятор, как раньше:

```
% javac Hello.java
```

Результатом будет файл *Hello.class*.

Для запуска программы следует, как прежде, вызвать интерпретатор Java, но на этот раз вам придется указать полностью квалифицированное имя программы, чтобы интерпретатору стало в точности известно, какую именно программу вы хотите исполнить:

```
% java com.davidflanagan.examples.basics.Hello
```

Установив переменную `CLASSPATH`, вы можете запускать интерпретатор Java из любого каталога вашей системы, и он всегда найдет нужную программу. Если набор вручную таких длинных имен утомляет, вы, возможно, изготовите для себя подходящий командный файл или сценарий, который будет выполнять это автоматически.

Заметьте, что таким же образом запускаются и исполняются все программы на Java, так что мы не будем повторять описание этих шагов. Один шаг, который вы сможете, разумеется, пропустить, – это набор программы. Вы можете скачать исходные коды примеров с сайта *http://www.davidflanagan.com/javaexamples2*.

FizzBuzz

Я научился играть в FizzBuzz очень давно, на уроках французского в начальной школе, где это помогало учиться счету по-французски. Игроки по очереди считают вслух от 1 и далее. Правила просты: на своем ходу вы называете очередное число. Но если это число кратно пяти, вместо него следует произнести слово «fizz» (желательно с французским акцентом). Если число кратно семи, надо сказать «buzz». А если число делится и на семь, и на пять, нужно произнести «fizzbuzz». Ошибившийся вылетает, и игра продолжается без него.

Пример 1.2 – это программа на Java под названием `FizzBuzz`, которая играет в эту игру. Конечно, это не самый интересный вариант игры, так как компьютер играет сам с собой и к тому же считает не по-французски! Нам будет интересен Java-код, содержащийся в этом примере. В нем демонстрируется применение цикла `for` для счета от 1 до 100 и используются операторы `if/else` для принятия решения о том, что следует вывести: число либо «fizz», либо «buzz», либо «fizzbuzz». (В этом примере оператор `if/else` появляется в форме оператора `if/elseif/elseif/else`, что мы вкратце обсудим позже.)

В этой программе используется метод `System.out.print()`. Он отличается от метода `System.out.println()` только тем, что не заканчивает выводимую строку. Все, что выводится далее, выводится в той же строке.

В этом примере показан еще один стиль представления комментариев. Любой набор строк между парой символов /* и парой символов */ является в Java комментарием и игнорируется компилятором. Когда комментарии начинаются с последовательности /**, как это имеет место в одном из комментариев примера, этот текст является еще и *doc-комментарием*. Это означает, что его содержимое используется программой *javadoc*, автоматически создающей API-документацию на основании исходных кодов Java.

Пример 1.2. FizzBuzz.java

```
package com.davidflanagan.examples.basics;
/**
 * Эта программа играет в FizzBuzz. Она считает до 100, заменяя каждое число,
 * кратное 5, словом «fizz», каждое число, кратное 7, – словом «buzz»
 * и каждое число, кратное 35, – словом «fizzbuzz». Для определения того,
 * делится ли одно число на другое, в ней используется
 * оператор остатка целочисленного деления (%).
 **/
public class FizzBuzz {                        // В Java все является классом
  public static void main(String[] args) {  // Каждая программа содержит main()
    for(int i = 1; i <= 100; i++) {          // Считаем от 1 до 100
      if (((i % 5) == 0) && ((i % 7) == 0)) // Делится ли число и на 5, и на 7?
        System.out.print("fizzbuzz");
      else if ((i % 5) == 0)                 // Делится ли число на 5?
        System.out.print("fizz");
      else if ((i % 7) == 0)                 // Делится ли число на 7?
        System.out.print("buzz");
      else System.out.print(i);             // Число не делится ни на 5, ни на 7
      System.out.print(" ");
    }
    System.out.println();
  }
}
```

Операторы `for` и `if/else` следует пояснить для тех программистов, которые с ними не встречались. Оператор `for` описывает цикл, в котором некоторый фрагмент кода выполняется несколько раз. За ключевым словом `for` следуют три выражения языка Java, определяющие параметры цикла. Синтаксис определения цикла таков:

```
for(инициализация ; проверка ; обновление)
тело
```

В выражении *инициализация* осуществляется необходимая инициализация. Оно исполняется только однажды, перед началом цикла. Здесь обычно устанавливается начальное значение переменной цикла. Часто, как и в этом примере, счетчик цикла используется только в цикле, так что в выражении *инициализация* переменная также и объявляется.

В выражении *проверка* проверяется, следует ли продолжать цикл. Это выражение вычисляется перед каждым исполнением тела цикла. Если его значением оказывается `true` (истинно), цикл исполняется. Если

же его значением оказывается false (ложно), тело цикла не исполняется, и цикл прекращается.

Выражение *обновление* вычисляется после каждой итерации цикла; оно осуществляет все необходимое для следующей итерации цикла. Обычно оно просто увеличивает или уменьшает значение счетчика цикла.

Наконец, *тело* – это код Java, исполняющийся в цикле. Он может представлять собой одиночный оператор Java или целый блок кода Java, заключенный в фигурные скобки.

Из этого объяснения становится ясно, что цикл for в примере 1.2 считает от 1 до 100.

Оператор if/else проще оператора for. Его синтаксис таков:

```
if (выражение)
    оператор1
else
    оператор2
```

Когда интерпретатор Java встречает оператор if, он вычисляет значение *выражения*. Если оно оказывается true, исполняется *оператор1*. В противном случае исполняется *оператор2*. Вот и все, что делает if/else; здесь нет никакого зацикливания, так что выполнение программы продолжается с первого оператора, следующего за if/else. Оператор else и следующий за ним *оператор2* совершенно необязательны. Если они отсутствуют и значение *выражения* оказывается равным false, оператор if не делает ничего. Операторы, следующие за ключевыми словами if и else, могут быть как одиночными операторами Java, так и целыми блоками кода Java, заключенными в фигурные скобки.

Что следует отметить в связи с операторами if/else (и операторами for, кстати говоря), так это то, что они могут содержать другие операторы, в частности другие операторы if/else. Именно так этот оператор используется в примере 1.2, где мы видим нечто похожее на оператор if/elseif/elseif/else. На самом деле это просто оператор if/else, находящийся внутри оператора if/else, заключенного в другом операторе if/else. Эта структура проясняется, если переписать код с применением фигурных скобок:

```
if (((i % 5) == 0)&& ((i % 7) == 0))
  System.out.print("fizzbuzz");
else {
  if ((i % 5) == 0)
    System.out.print("fizz");
  else {
    if ((i % 7) == 0)
      System.out.print("buzz");
    else
      System.out.print(i);
  }
}
```

Заметим, однако, что подобные вложенные операторы `if/else` обычно записываются без таких фигурных скобок. Программная конструкция `else if` – это широко используемый прием, к которому вы скоро привыкнете. Вы, возможно, заметили также, что я придерживаюсь компактного стиля при программировании, стараясь по возможности все помещать в одной строке. Например, вам часто придется видеть следующее:

```
if (выражение) оператор
```

В таком виде программа компактнее и удобнее в обращении, поэтому ее легче изучать в напечатанном виде. В своих кодах вы, возможно, предпочтете более структурированный, менее компактный стиль.

Числа Фибоначчи

Числа Фибоначчи – это последовательность чисел, в которой каждое следующее число равно сумме двух предыдущих. Начало этой последовательности – 1, 1, 2, 3, 5, 8, 13, и ее можно бесконечно продолжать. Эта последовательность встречается в природе в самых неожиданных местах. Например, число лепестков у большинства видов цветов является одним из чисел Фибоначчи.

Пример 1.3 представляет собой программу, которая вычисляет и отображает 20 первых чисел ряда Фибоначчи. По поводу этой программы следует сказать несколько слов. Во-первых, в ней снова используется оператор `for`. В нем также объявляются и используются переменные для хранения двух последних чисел последовательности, так что их можно сложить, чтобы получить следующее число.

Пример 1.3. Fibonacci.java

```
package com.davidflanagan.examples.basics;
/**
 * Эта программа распечатывает 20 первых чисел ряда Фибоначчи.
 * Каждый его элемент формируется как сумма двух предыдущих элементов ряда,
 * начиная с элементов 1 и 1.
 **/
public class Fibonacci {
 public static void main(String[] args) {
  int n0 = 1, n1 = 1, n2;        // Инициализация переменных
  System.out.print(n0 + " " +    // Печать первого и второго элементов
       n1 + " ");                // ряда
  for(int i = 0; i < 18; i++) { // Цикл для следующих 18 элементов
   n2 = n1 + n0;           // Следующий элемент является суммой двух предыдущих
   System.out.print(n2 + " ");  // Распечатать
   n0 = n1;       // Первый с конца становится вторым с конца
   n1 = n2;       // а текущее число становится последним
  }
  System.out.println();          // Завершаем строку
 }
}
```

Использование аргументов командной строки

Как мы уже видели, всякая самостоятельная программа на Java должна содержать объявление метода, в точности соответствующее следующей сигнатуре:

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

Эта сигнатура говорит о том, что методу `main()` передается массив строк. Что это за строки и откуда они берутся? Массив `args` содержит все аргументы, передаваемые интерпретатору Java в командной строке вслед за именем запускаемого на выполнение класса. Пример 1.4 представляет собой программу `Echo`, читающую и распечатывающую эти аргументы. Вы могли бы вызвать эту программу так:

```
% java com.davidflanagan.examples.basics.Echo this is a test
```

Программа отвечает:

```
this is a test
```

В этом случае длина массива `args` равна четырем. Первый элемент массива `args[0]` – это строка «this», а последний элемент массива `args[3]` – это строка «test». Как видите, массивы в Java начинаются с элемента `0`. Если вы до этого программировали на языке, в котором нумерация элементов массивов начинается с 1, вам придется как-то к этому привыкнуть. В частности, вы должны запомнить, что если длина массива `a` равна `n`, последним элементом массива будет `a[n-1]`. Вы можете определить длину массива, приписав к его имени `.length`, как это показано в примере 1.4. Этот пример демонстрирует также применение цикла `while`. Цикл `while` – это упрощенная форма цикла `for`; в нем вам придется самостоятельно производить инициализацию и обновление счетчика. Большинство циклов `for` могут быть переписаны как циклы `while`, но компактный синтаксис цикла `for` делает его более распространенным. В этом примере цикл `for` вполне подошел бы и был бы даже предпочтительнее.

Пример 1.4. Echo.java

```
package com.davidflanagan.examples.basics;
/**
 * Эта программа распечатывает все заданные в ее командной строке аргументы.
 **/
public class Echo {
 public static void main(String[] args) {
  int i = 0;                          // Инициализация переменной цикла
  while(i < args.length) {            // Цикл до конца массива
   System.out.print(args[i] + " ");   // Печать каждого из аргументов
   i++;                               // Увеличение переменной цикла
  }
```

```
    System.out.println();              // Переход на следующую строку
  }
}
```

Эхо-вывод в обратном порядке

Пример 1.5 очень похож на программу `Echo` из примера 1.4 за исключением того, что в нем аргументы командной строки выводятся в обратном порядке, как и символы в каждом из аргументов. Таким образом, программа `Reverse`, вызванная способом, представленным ниже, произведет следующий вывод:

```
% java com.davidflanagan.examples.basics.Reverse this is a test
tset a si siht
```

Эта программа интересна тем, что ее вложенные циклы `for` считают не вперед, а назад. Интересно в ней также то, что она манипулирует строковыми объектами (`String`), вызывая методы этих объектов, и синтаксис становится несколько сложнее. Рассмотрим в качестве примера центральное выражение из этой программы:

```
args[i].charAt(j).
```

Это выражение первым делом выделяет i-й элемент массива `args[]`. Из объявления массива в сигнатуре метода `main()` мы знаем, что это массив строк (`String` array) и что он, следовательно, состоит из строковых объектов (`String` objects). (В языке Java `String` это не примитивный тип, такой как целые и логические значения: строки являются полноценными объектами.) Выделив из массива i-ю строку, мы вызываем принадлежащий этому объекту метод `charAt()` с аргументом `j`. (Символ «`.`» в этом выражении указывает на то, что вы имеете дело с методом или с полем объекта.) Как можно догадаться из названия метода (и при желании убедиться, заглянув в справочное руководство), он выделяет указанный символ из объекта `String`. Таким образом, это выражение извлекает j-й символ из i-го аргумента командной строки. Вооружившись этими знаниями, вы сумеете до конца разобраться в примере 1.5.

Пример 1.5. Reverse.java

```
package com.davidflanagan.examples.basics;
/**
 * Эта программа выводит «от конца к началу» аргументы, заданные
 * в командной строке.
 **/
public class Reverse {
 publicstatic void main(String[] args) {
  // Цикл проходит массив аргументов от конца к началу
  for(int i = args.length-1; i >= 0; i--) {
   // Цикл проходит от конца к началу символы в каждом аргументе
   for(int j=args[i].length()-1; j>=0; j--) {
    // Печатается символ j аргумента i.
```

```
      System.out.print(args[i].charAt(j));
     }
     System.out.print(" "); // После каждого аргумента выводится пробел
    }
    System.out.println();  // И, закончив, переходим на следующую строку
   }
  }
```

FizzBuzz с оператором switch

Пример 1.6 представляет другую версию игры `FizzBuzz`. В ней для определения того, что следует выводить для каждого числа, вместо вложенных операторов `if/else` используется оператор `switch`. Сначала посмотрите на пример, а затем прочитайте объяснение того, как работает оператор `switch`.

Пример 1.6. FizzBuzz2.java

```
package com.davidflanagan.examples.basics;
/**
 * Этот класс очень похож на класс FizzBuzz, но в нем вместо
 * повторяющихся операторов if/else применяется оператор switch
 **/
public class FizzBuzz2 {
 public static void main(String[] args) {
  for(int i = 1; i <= 100; i++) {  // Считаем от 1 до 100
   switch(i % 35) {                // Каков остаток при делении на 35?
   case 0:                         // Для кратных 35...
    System.out.print("fizzbuzz "); // Печать «fizzbuzz»
    break;                         // Не пропустите этот оператор!
   case 5: case 10: case 15:       // Если остаток - один из представленных,
   case 20: case 25: case 30:      // число кратно 5,
    System.out.print("fizz ");     // и печатается «fizz»
    break;
   case 7: case 14: case 21: case 28: // Для всех кратных 7...
    System.out.print("buzz ");        // печатается «buzz»
    break;
   default:                      // Для любого другого числа...
    System.out.print(i + " ");   // печатается число
    break;
   }
  }
  System.out.println();
 }
}
```

Оператор `switch` работает как диспетчер на маневровой площадке, из множества вариантов выбирающий для поезда (программы) подходящий путь (фрагмент кода). Оператор `switch` часто может служить альтернативой повторяющимся операторам `if/else`, но он работает, только когда тестируемое значение принадлежит примитивному целочис-

ленному типу (другими словами, если оно не `double` и не `String`) и когда его значение сравнивается с константами. Синтаксис оператора `switch` таков:

```
switch(выражение) {
 операторы
}
```

За оператором `switch` следуют *выражение* в скобках и блок кода в фигурных скобках. После вычисления *выражения* оператор `switch` исполняет определенную часть кода в скобках, зависящую от значения выражения. Как оператор `switch` узнает, где начинается код, соответствующий определенному значению? Эта информация обозначается метками `case:` и специальной меткой `default:`. За каждой меткой `case:` следует некоторое целочисленное значение. Если результат вычисления *выражения* оказывается равным этому значению, оператор `switch` начинает исполнять код, непосредственно следующий за меткой `case:`. Если не найдена метка `case:`, соответствующая значению *выражения*, оператор `switch` исполняет код, следующий за меткой `default:`, если таковая имеется. Если метки `default:` нет, `switch` не делает ничего.

Оператор `switch` необычен тем, что в нем меткам `case:` не соответствуют блоки кода. Вместо этого метки `case:` и `default:` просто отмечают точки входа в один большой блок кода. Обычно за каждой меткой следуют несколько операторов и затем оператор `break`, вызывающий передачу управления за пределы оператора `switch`. Если вы пропустите оператор `break` в конце кода, соответствующего метке, исполнение «провалится» в код, относящийся к следующей метке. Если вы хотите посмотреть на это в действии, уберите оператор `break` из примера 1.6 и посмотрите, что получится, когда вы запустите программу. Пропуск оператора `break` в операторе `switch` – распространенный источник ошибок.

Вычисление факториалов

Факториал целого числа получается как произведение этого числа на все положительные целые, которые меньше него. Таким образом, факториал 5 (записывается `5!`) – это произведение `5*4*3*2*1`, или `120`. Пример 1.7 определяет класс `Factorial`, содержащий метод `factorial()`, вычисляющий факториалы. Этот класс не является самостоятельной программой, но определенный в нем метод может использоваться любой другой программой. Сам по себе метод очень прост; в следующих разделах мы увидим несколько его вариантов. В качестве упражнения вы могли бы придумать, как переписать этот пример с использованием цикла `while` вместо цикла `for`.

Пример 1.7. Factorial.java

```
package com.davidflanagan.examples.basics;
/**
```

```
 * Этот класс не определяет метод main() и поэтому не является
 * самостоятельной программой. Он, тем не менее, определяет полезный
 * метод, который можно использовать в других программах.
 **/
public class Factorial {
 /** Вычисляем и возвращаем x!, факториал x */
 public static int factorial(int x) {
  if (x < 0) throw new IllegalArgumentException("x должен быть >= 0");
  int fact = 1;
  for(int i = 2; i <= x; i++) // Цикл
   fact *= i;                 // Краткая запись для fact = fact * i;
  return fact;
 }
}
```

Рекурсивные факториалы

Пример 1.8 демонстрирует еще один способ вычисления факториалов. В нем используется прием программирования, называемый *рекурсией*. Рекурсия имеет место, когда метод вызывает сам себя, или, другими словами, рекурсивно обращается к себе. Рекурсивный алгоритм вычисления факториалов основывается на том факте, что `n!` равняется `n*(n-1)!`. Такой способ вычисления факториалов является классическим примером рекурсии. В этом случае этот прием не особенно эффективен, но у рекурсии есть множество важных применений, и этот пример доказывает, что ее применение в Java совершенно законно. В этом примере также вместо типа данных `int`, являющегося 32-битовым целым, используется тип данных `long`, являющийся 64-битовым целым. Факториалы быстро становятся очень большими, так что дополнительная емкость типа `long` делает метод `factorial()` более полезным.

Пример 1.8. Factorial2.java

```
package com.davidflanagan.examples.basics;
/**
 * Этот класс демонстрирует рекурсивный способ вычисления факториалов. Этот
 * метод многократно вызывает сам себя, основываясь на формуле: n! = n * (n-1)!
 **/
public class Factorial2 {
 public static long factorial(long x) {
  if (x < 0) throw new IllegalArgumentException("x должен быть >= 0");
  if (x <= 1) return 1;           // Здесь рекурсия прекращается
  else return x * factorial(x-1); // Шаг рекурсии - вызов самого себя
 }
}
```

Кэширование факториалов

Пример 1.9 предъявляет усовершенствованный вариант предыдущих примеров, вычисляющих факториалы. Факториалы – это идеальные кандидаты для кэширования, поскольку их вычисление требует времени и, что еще важнее, вы можете вычислить не так уж много факториалов в силу ограниченности типа данных `long`. Поэтому в этих примерах, раз уж факториал вычислен, его значение сохраняется для будущего употребления.

Помимо демонстрации приемов кэширования в этом примере вводится кое-что новое. Во-первых, в классе `Factorial3` определяются статические поля:

```
static long[] table = new long[21];
static int last = 0;
```

Статические поля – это род переменных, сохраняющих, однако, свои значения в промежутках между вызовами метода `factorial()`. Это означает, что в статических полях могут сохраняться значения, вычисленные при одном вызове метода, для использования при другом его вызове.

Во-вторых, в этом примере можно увидеть, как создаются массивы:

```
static long[] table = new long[21];
```

Первая половина этой строки (перед знаком =) объявляет статическое поле `table`, которое будет массивом значений типа `long`. Вторая половина этой строки создает массив из 21 значения типа `long` при помощи оператора `new`.

Наконец, в этом примере показано, как генерируются исключения:

```
throw new IllegalArgumentException("Переполнение: x слишком велик.");
```

Исключения – это вид объектов Java. Они, как и массивы, создаются при помощи ключевого слова `new`. Если программа генерирует объект-исключение при помощи оператора `throw`, это означает, что возникло неожиданное обстоятельство или ошибка. Когда возникает исключение, управление в программе передается ближайшей секции `catch` оператора `try/catch`. Эта секция должна содержать код для обработки исключений. Если исключение не будет перехвачено, исполнение программы прекращается с сообщением об ошибке.

В примере 1.9 возникает исключение, уведомляющее вызывающую процедуру о том, что переданный ею аргумент слишком велик или слишком мал. Аргумент слишком велик, если он больше 20, поскольку мы не можем вычислять факториалы, превышающие `20!`. Аргумент слишком мал, если он меньше 0, так как факториалы определены только для неотрицательных целых чисел. Как перехватывать и обрабатывать исключения, показано в следующих примерах этой главы.

Пример 1.9. Factorial3.java

```
package com.davidflanagan.examples.basics;
/**
 * Этот класс вычисляет факториалы и кэширует результаты в таблице
 * для дальнейшего употребления. 20! - самый большой факториал,
 * который мы можем вычислить с применением типа данных long,
 * поэтому проверим аргумент и выдадим исключение, если аргумент
 * окажется слишком большим или слишком маленьким.
 **/
public class Factorial3 {
 // Создаем массив для хранения факториалов от 0! до 20!.
 static long[] table = new long[21];
 // «Статический инициализатор»: инициализируем первый элемент массива
 static { table[0] = 1; } // Факториал 0 равен 1.
 // Запоминаем номер последнего вычисленного факториала
 static int last = 0;
 public static long factorial(int x) throws IllegalArgumentException {
  // Проверяем, не слишком ли мал или велик x. Выдаем исключение,
  // если это оказывается так
  if (x >= table.length) // ".length" возвращает длину любого массива
   throw new IllegalArgumentException("Переполнение: x слишком велик.");
  if (x<0) throw new IllegalArgumentException("x должен быть неотрицательным.");
  // Вычисляем и кэшируем все пока еще несохраненные значения
  while(last < x) {
   table[last + 1] = table[last] * (last + 1);
   last++;
  }
  // Возвращаем кэшированный факториал x
  return table[x];
 }
}
```

Вычисление факториалов больших чисел

В предыдущем разделе мы узнали, что `20!` – это самый большой факториал, помещающийся в 64-битовом целом. Но что делать, если нужно вычислить `50!` или `100!`? Класс `java.math.BigInteger` представляет сколь угодно большие целые числа и обеспечивает методы для арифметических операций над ними. Пример 1.10 использует класс `BigInteger` для вычисления факториалов любого размера. Он содержит также простой метод `main()`, представляющий собой самостоятельную программу для тестирования метода `factorial()`. Эта тестовая программа может, например, сообщить, что `50!` – это следующее 65-значное число:

```
30414093201713378043612608166064768844377641568960512000000000000
```

Пример 1.10 знакомит вас с оператором `import`. Этот оператор появляется в начале файла Java до определения какого-либо класса (но после объявления `package`). Он позволяет сообщить компилятору, какие

классы используются в программе. После того как, например, класс `java.math.BigInteger` импортирован, вам уже не нужно писать его полное имя; вы можете ссылаться на него просто как на `BigInteger`. Вы можете импортировать также целый пакет классов, включив в файл, например, такую строку:

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

Обратите внимание на то, что классы пакета `java.lang` автоматически импортированы, поскольку они принадлежат текущему пакету, которым в нашем случае является `com.davidflanagan.examples.basics`.

В примере 1.10 применяется та же техника кэширования, что и в примере 1.9. Однако поскольку множество факториалов, которые могут быть вычислены, теперь не ограничено сверху, для кэширования нельзя применить массив фиксированного размера. Вместо этого в примере используется `java.util.ArrayList` — служебный класс, реализующий структуру данных, подобную массивам и способную разрастаться до любого нужного размера. Поскольку `ArrayList` — это объект, а не массив, при работе с ним приходится применять методы, такие как `size()`, `add()` и `get()`. Точно так же `BigInteger` — это объект, а не примитивное значение, поэтому для умножения объектов `BigInteger` нельзя применять оператор `*`. Вместо этого используется метод `multiply()`.

Пример 1.10. Factorial4.java

```
package com.davidflanagan.examples.basics;
// Импортируем классы, которые намереваемся использовать в нашей программе.
// Импортировав класс, мы уже не обязаны называть его полным именем.
import java.math.BigInteger; // Импортируем класс BigInteger из пакета java.math
import java.util.*;          // Импортируем все классы (включая ArrayList)
                             // пакета java.util
/**
 * В этой версии программы применяются целые числа произвольно большого
 * размера, поэтому вычисляемые значения не ограничены сверху.
 * Для кэширования вычисленных значений в ней применяется объект ArrayList
 * вместо массивов фиксированного размера. ArrayList похож на массив,
 * но может разрастаться до любого размера. Метод factorial() объявлен
 * как «synchronized», поэтому его можно смело использовать в
 * многопоточных программах. При изучении этого класса познакомьтесь
 * с java.math.BigInteger и java.util.ArrayList. Работая с версиями Java,
 * предшествующими Java 1.2, используйте Vector вместо ArrayList.
 **/
public class Factorial4 {
 protected static ArrayList table = new ArrayList(); // Создаем кэш
 static {  // Первый элемент кэша инициализируется значением 0! = 1
 table.add(BigInteger.valueOf(1));
 }
 /** Метод factorial(), использующий объекты BigInteger,
     сохраняемые в ArrayList */
 public static synchronized BigInteger factorial(int x) {
```

```
 if (x<0) throw new IllegalArgumentException("x должен быть неотрицательным.");
 for(int size = table.size(); size <= x; size++) {
  BigInteger lastfact = (BigInteger)table.get(size-1);
  BigInteger nextfact = lastfact.multiply(BigInteger.valueOf(size));
  table.add(nextfact);
 }
 return (BigInteger) table.get(x);
 }
 /**
 * Простой метод main(), который можно использовать в качестве
 * самостоятельной тестирующей программы для нашего метода factorial().
 **/
 public static void main(String[] args) {
 for(int i = 0; i <= 50; i++)
  System.out.println(i + "! = " + factorial(i));
 }
}
```

Обработка исключений

Пример 1.11 демонстрирует программу, использующую метод `Integer.parseInt()` для преобразования в число строки символов, заданной в командной строке. Затем программа вычисляет и печатает факториал этого числа, используя метод `Factorial4.factorial()`, определенный в примере 1.10. Пока все очень просто; это потребовало всего две строки кода. Оставшаяся часть примера занимается обработкой исключений, другими словами, заботится обо всем, что может пойти не так, как надо. Для обработки исключений в Java используется оператор `try/catch`. В его секции `try` содержится блок кода, который может выдать исключение. За ним может следовать любое число секций `catch`. Код в каждой секции `catch` заботится об определенном типе исключений.

В примере 1.11 возможны три ошибки пользовательского ввода, способные помешать нормальному выполнению программы. Поэтому две главные строки кода программы «обернуты» в секцию `try`, за которой следуют три секции `catch`. Каждая из последних уведомляет пользователя об определенной ошибке, печатая соответствующее сообщение. Этот пример очень прозрачен. Вы можете обратиться к главе 2 в «Java in a Nutshell», где исключения объясняются более подробно.

Пример 1.11. FactComputer.java

```
package com.davidflanagan.examples.basics;
/**
 * Эта программа вычисляет и отображает факториал числа,
 * заданного в командной строке. При помощи оператора try/catch
 * в ней обрабатываются возможные ошибки пользовательского ввода.
 **/
public class FactComputer {
 public static void main(String[] args) {
```

```
// Пробуем вычислить факториал. Если что-то оказывается не в порядке,
// обрабатываем исключения при помощи секции catch.
try {
 int x = Integer.parseInt(args[0]);
 System.out.println(x + "! = " + Factorial4.factorial(x));
}
// Пользователь забыл задать аргумент.
// Исключение возникает, если args[0] остается неопределенным.
catch (ArrayIndexOutOfBoundsException e) {
 System.out.println("Вы должны задать аргумент");
 System.out.println("Формат: java FactComputer <число>");
}
// Аргумент не является числом. Выдается методом Integer.parseInt().
catch (NumberFormatException e) {
 System.out.println("Задаваемый аргумент должен быть целым числом");
}
// Аргумент < 0. Выдается методом Factorial4.factorial()
catch (IllegalArgumentException e) {
 // Отображает сообщение, выданное методом factorial():
 System.out.println("Плохой аргумент: " + e.getMessage());
}
}
}
```

Интерактивный ввод

Пример 1.12 показывает еще одну программу для вычисления факториалов. В отличие от примера 1.11, однако, эта программа не останавливается после вычисления одного факториала. Вместо этого она предлагает пользователю ввести число, считывает это число, печатает его факториал и затем возвращается в начало и предлагает пользователю ввести новое число. Интереснее всего в этой программе техника, применяемая для считывания пользовательского ввода с клавиатуры. Для этого в ней используется метод `readLine()` объекта `BufferedReader`. Строка, в которой создается `BufferedReader`, может показаться запутанной. Примите пока на веру, что это работает; до главы 3 вам необязательно понимать, как это работает. В примере 1.12 следует также обратить внимание на применение метода `equals()` к объекту `line` типа `String` для проверки, не набрал ли пользователь «quit».

Код для анализа пользовательского ввода, вычисления и печати факториала такой же, как в примере 1.11, и снова он заключен в секцию `try`. В примере 1.12, однако, есть только одна секция `catch` для обработки возможных исключений. Она обрабатывает все объекты исключений, принадлежащих типу `Exception`. `Exception` — это суперкласс исключений всех типов, поэтому вызывается одна секция `catch` вне зависимости от типа возникшего исключения.

Пример 1.12. FactQuoter.java

```
package com.davidflanagan.examples.basics;
import java.io.*; // Импортируем все классы пакета java.io.,
                  // избавляя, тем самым, себя от лишнего набора.
/**
 * Эта программа отображает факториалы чисел,
 * вводимых пользователем в интерактивном режиме
 **/
public class FactQuoter {
 public static void main(String[] args) throws IOException {
 // Так подготавливается чтение строк, вводимых пользователем
 BufferedReader in=new BufferedReader(new InputStreamReader(System.in));
 // Бесконечный цикл
 for(;;) {
  // Отображается подсказка для пользователя
  System.out.print("FactQuoter> ");
  // Считывается введенная пользователем строка
  String line = in.readLine();
  // Если считан конец файла,
  // или если пользователь набрал «quit», то конец
  if ((line == null) || line.equals("quit")) break;
  // Пытаемся проанализировать строку, вычислить и напечатать факториал
  try {
  int x = Integer.parseInt(line);
  System.out.println(x + "! = " + Factorial4.factorial(x));
  }
  // Если что-то не в порядке, отображается общее сообщение об ошибке
  catch(Exception e) { System.out.println("Invalid Input"); }
 }
 }
}
```

Применение объекта StringBuffer

Относительно класса `String`, используемого в Java для представления строк, следует отметить тот факт, что его объекты неизменяемы. Другими словами, не существует методов, которые позволяют изменить содержимое строки. Работающие со строками методы возвращают новую строку, а не преобразованную копию старой. Если вы хотите отредактировать строку по месту, вам придется воспользоваться объектом `StringBuffer`.

Пример 1.13 демонстрирует применение объекта `StringBuffer`. В этом примере в интерактивном режиме считывается строка пользовательского ввода, как в примере 1.12, и для хранения строки создается объект `StringBuffer`. Затем, применяя подстановочный шифр *rot13*, который просто «сдвигает по кругу» каждую букву алфавита на 13 мест (за Z следует A), программа кодирует каждый символ строки. За счет использования объекта `StringBuffer` можно поочередно заменять все сим-

волы строки. Сеанс с программой `Rot13Input` может выглядеть примерно так:

```
% java com.davidflanagan.examples.basics.Rot13Input
> Hello there. Testing, testing!
Uryyb gurer. Grfgvat, grfgvat!
> quit
%
```

Метод `main()` из примера 1.13 вызывает другой метод, `rot13()`, для осуществления непосредственно шифрования символа. Этот метод демонстрирует применение примитивного типа `char` языка Java и символьных литералов (то есть символов, заключенных в апострофы и используемых в программе буквально).

Пример 1.13. Rot13Input.java

```
package com.davidflanagan.examples.basics;
import java.io.*; // Мы будем вводить данные, поэтому импортируем
                  // классы ввода/вывода
/**
 * Эта программа читает вводимые пользователем строки, кодирует их
 * при помощи тривиального подстановочного шифра «Rot13»
 * и затем печатает закодированную строку.
 **/
public class Rot13Input {
 public static void main(String[] args) throws IOException {
 // Подготовка к чтению вводимых пользователем строк
 BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
 for(;;) {                       // Бесконечный цикл
  System.out.print("> ");        // Печать подсказки
  String line = in.readLine();   // Чтение строки
  if ((line == null) || line.equals("quit"))  // Если конец файла или «quit»...
  break;                                      // ...выход из цикла
  StringBuffer buf = new StringBuffer(line); // Применение StringBuffer
  for(int i = 0; i < buf.length(); i++)      // Для каждого символа...
  buf.setCharAt(i, rot13(buf.charAt(i)));    // ...прочитать, закодировать,
                                             // сохранить
  System.out.println(buf);          // Печатается закодированная строка
 }
 }

/**
 * Этот метод осуществляет подстановочное шифрование Rot13.
 * Он «сдвигает по кругу» каждую букву алфавита на 13 мест. Поскольку
 * в латинском алфавите 26 букв, этот метод и кодирует, и декодирует.
 **/
 public static char rot13(char c) {
 if ((c >= 'A') && (c <= 'Z')) { // Для заглавных букв
  c += 13;                       // Сдвигает вперед на 13
  if (c > 'Z') c -= 26;          // и вычитает при необходимости 26
 }
 if ((c >= 'a') && (c <= 'z')) { // То же самое для строчных букв
```

```
    c += 13;
    if (c > 'z') c -= 26;
   }
   return c;                          // Возвращает преобразованную букву
   }
}
```

Сортировка чисел

Пример 1.14 реализует простой (но неэффективный) алгоритм сортировки массива чисел. В этом примере не вводятся новые элементы синтаксиса Java, но он интересен, поскольку в нем достигается уровень сложности, свойственный реальным программам. Алгоритм сортировки манипулирует элементами массива с применением оператора `if` в цикле `for`, заключенном в другой цикл `for`. Не пожалейте времени на внимательное изучение этого примера. Убедитесь, что вы точно поняли, как он сортирует массив чисел.

Пример 1.14. SortNumbers.java

```
package com.davidflanagan.examples.basics;
/**
 * Этот класс демонстрирует сортировку чисел при помощи простого алгоритма
 **/
public class SortNumbers {
   /**
   * Это очень простой алгоритм сортировки, не очень эффективный
   * при сортировке больших наборов элементов
   **/
   public static void sort(double[] nums) {
     // Цикл по всем элементам массива, в ходе которого
     // осуществляется сортировка.
     // На каждом шаге среди оставшихся неотсортированными
     // отыскиваем наименьший элемент и перемещаем
     // на первую неотсортированную позицию в массиве.
     for(int i = 0; i < nums.length; i++) {
         int min = i; // Хранит индекс наименьшего элемента
         // Находим наименьший элемент от i до конца массива
         for(int j = i; j < nums.length; j++) {
             if (nums[j] < nums[min]) min = j;
         }
         // Меняем местами наименьший элемент с элементом i.
         // Элементы между 0 и i остаются при этом отсортированными.
         double tmp;
         tmp = nums[i];
         nums[i] = nums[min];
         nums[min] = tmp;
     }
 }
```

```
 /** Это простая тестирующая программа для вышеприведенного алгоритма */
 public static void main(String[] args) {
     double[] nums = new double[10];    // Создается массив для хранения чисел
     for(int i = 0; i < nums.length; i++) // Генерируются случайные числа
         nums[i] = Math.random() * 100;
     sort(nums);  // Они сортируются
     for(int i = 0; i < nums.length; i++) // и распечатываются
         System.out.println(nums[i]);
     }
}
```

Вычисление простых чисел

В примере 1.15 с применением алгоритма «Решето Эратосфена» вычисляется наибольшее простое число, не превосходящее заданное значение. Этот алгоритм находит простые числа путем исключения всех чисел, кратных меньшим простым. Как и в примере 1.14, здесь нет новых синтаксических конструкций языка Java, но этой красивой и нетривиальной программой приятно закончить настоящую главу. Эта программа может показаться обманчиво простой, но в ней много чего происходит, так что убедитесь, что вы хорошо поняли, как она справляется с простыми числами.

Пример 1.15. Sieve.java

```
package com.davidflanagan.examples.basics;
/**
 * Эта программа вычисляет простые числа, применяя алгоритм
 * «Решето Эратосфена»: уберите числа, кратные меньшим простым числам,
 * и все оставшиеся будут простыми. Она печатает наибольшее число,
 * не превосходящее аргумент, заданный в командной строке.
 **/
public class Sieve {
 public static void main(String[] args) {
     // Мы вычислим все простые числа, не превосходящие заданное значение, или,
     // если аргумент не указан, все простые числа, не превосходящие 100.
     int max = 100;  // Присваиваем значение, принимаемое по умолчанию
     try { max = Integer.parseInt(args[0]); } // Анализируем заданный
                                              // пользователем аргумент
     catch (Exception e) {}  // Молча игнорируем исключения
     // Создаем массив, где для каждого числа указано, простое оно или нет.
     boolean[] isprime = new boolean[max+1];
     // Предполагаем, что все числа простые, пока не доказано обратное.
     for(int i = 0; i <= max; i++) isprime[i] = true;
     // Мы, однако, знаем, что 0 и 1 - не простые числа.
     // Обратите на это внимание.
     isprime[0] = isprime[1] = false;
     // Чтобы вычислить все простые числа меньше max, нужно убрать
     // числа, кратные всем целым, меньшим, чем квадратный корень из max.
     int n = (int) Math.ceil(Math.sqrt(max)); // См. класс java.lang.Math
     // Теперь для каждого целого i от 0 до n:
```

```
        //     Если i простое число, тогда никакое из кратных ему не простое,
        //     отметим это в нашем массиве. Если i не простое число, кратные
        //     ему уже удалены (в качестве кратных его простому делителю),
        //     значит, этот случай пропускаем.
        for(int i = 0; i <= n; i++) {
            if (isprime[i])                          // Если i - простое число,
                 for(int j = 2*i; j <= max; j = j + i) // цикл по кратным,
                    isprime[j] = false;              // они не являются простыми.
        }
        // Теперь найдем наибольшее простое:
        int largest;
        for(largest = max; !isprime[largest]; largest--) ; // Пустое тело цикла
        // Выведем результат
       System.out.println("Наибольшее простое число, не превосходящее " + max +
        " это " + largest);
    }
}
```

Упражнения

1-1. Напишите программу, которая считает от 1 до 15, печатает каждое число и затем считает двойками в обратном направлении до 1, снова печатая каждое число.

1-2. Каждый элемент в ряде Фибоначчи получается сложением двух предыдущих. Какой ряд получится, если складывать три предыдущих числа? Напишите программу, печатающую первые 20 элементов такого ряда.

1-3. Напишите программу, получающую в качестве аргументов в командной строке два числа и строку и печатающую подстроку заданной строки, определенную заданными числами. Например:

```
% java Substring hello 1 3
```

должна вывести: `ell`

Обработайте всевозможные исключения, обусловленные неправильным вводом.

1-4. Напишите программу, которая считывает в интерактивном режиме вводимые пользователем строки и печатает их в обратном порядке. Исполнение программы прекращается, когда пользователь введет «tiuq».

1-5. Класс `SortNumbers` показывает, как можно отсортировать массив чисел типа `double`. Напишите программу, применяющую этот класс для сортировки 100 чисел с плавающей точкой. Затем в интерактивном режиме предложите пользователю ввести число и отобразите соседние с ним в массиве большее и меньшее числа. Чтобы найти нужную позицию в отсортированном массиве, примените эффективный алгоритм двоичного поиска.

Глава 2

Объекты, классы и интерфейсы

Эта глава содержит примеры, подчеркивающие объектно-ориентированную природу Java. Она создавалась для совместного изучения с главой 3 книги «Java in a Nutshell». В последней предлагается подробное введение в понятия объектно-ориентированного программирования и описывается синтаксис, которым нужно владеть, чтобы программировать на Java. В нескольких следующих абзацах содержится сжатое описание объектно-ориентированной терминологии Java.

Объект – это набор значений, или *полей* (*fields*), а также *методов* (*methods*), оперирующих этими данными. Тип данных объекта называется *классом* (*class*); об объекте часто говорят как об *экземпляре* (*instance*) его класса. Класс определяет тип каждого поля объекта и предоставляет методы, оперирующие с данными, содержащимися в экземпляре класса. Объект создается при помощи оператора `new`, вызывающего для инициализации объекта *конструктор* (*constructor*) этого класса. Содержимое полей и методы объекта получаются и вызываются, соответственно, при помощи оператора «`.`» (точка).

Методы, оперирующие с полями объекта, называются *методами экземпляра*. Они отличаются от статических методов, или методов класса, которые мы видели в главе 1. Методы класса объявляются как `static`; они оперируют с самим классом, а не с индивидуальными экземплярами класса. Поля, принадлежащие классу, также могут объявляться как `static`, что делает их полями класса, а не полями экземпляра. В то время как каждый объект имеет копии всех полей экземпляра, есть только одна копия поля класса, и она совместно используется всеми экземплярами класса.

Поля и методы класса могут иметь различные уровни видимости – `public`, `private` и `protected`. Эти различные уровни видимости определяют

возможность использования полей и методов в различных контекстах. У каждого класса есть свой *базовый* (*родительский*) *класс*, от которого он *наследует* свои поля и методы. Класс, наследующий от другого класса, называется *подклассом* этого класса. Классы в Java образуют *иерархию классов*. Класс `java.lang.Object` – это корень иерархии; `Object` – это первичный базовый класс для всех классов в Java.

Интерфейс – это конструкция Java, определяющая, подобно классам, методы, но не предоставляющая никакой реализации этих методов. Класс может *реализовывать* (*implement*) интерфейс путем реализации каждого метода в интерфейсе.

Класс прямоугольника

Пример 2.1 демонстрирует класс, представляющий прямоугольник. Каждый экземпляр этого класса `Rect` имеет четыре поля, `x1`, `y1`, `x2` и `y2`, определяющих координаты углов прямоугольника. Класс `Rect` определяет также множество методов, оперирующих этими координатами.

Обратите внимание на метод `toString()`. Этот метод замещает (override) метод `toString()` класса `java.lang.Object`, служащего неявным (implicit) базовым классом класса `Rect`. Метод `toString()` создает объект класса `String` (строку), представляющий объект класса `Rect`. Как мы увидим, этот метод очень полезен при печати значений `Rect`.

Пример 2.1. Rect.java

```
package com.davidflanagan.examples.classes;
/**
 * Этот класс представляет прямоугольник. Его поля представляют координаты
 * углов этого прямоугольника. Его методы определяют операции, которые
 * могут осуществляться с объектами Rect.
 **/
public class Rect {
    // Это поля данных класса
    public int x1, y1, x2, y2;
    /**
     * Метод Rect - главный конструктор класса. Он просто использует свои
     * аргументы для инициализации полей нового объекта. Обратите внимание
     * на то, что его имя совпадает с именем класса и что в его сигнатуре
     * нет возвращаемого значения.
     **/
    public Rect(int x1, int y1, int x2, int y2) {
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
    }
    /**
     * Это еще один конструктор. Он определяется через предыдущий конструктор
     **/
```

```
    public Rect(int width, int height) { this(0, 0, width, height); }

    /** Это еще один конструктор. */
    public Rect() { this(0, 0, 0, 0); }
    /** Перемещение прямоугольника на указанные расстояния */
    public void move(int deltax, int deltay) {
        x1 += deltax; x2 += deltax;
        y1 += deltay; y2 += deltay;
    }
    /** Проверяем, находится ли заданная точка внутри прямоугольника */
    public boolean isInside(int x, int y) {
        return ((x >= x1) && (x <= x2) && (y >= y1) && (y <= y2));
    }
    /**
     * Возвращается объединение этого прямоугольника с другим. Другими
словами,
     * возвращается наименьший прямоугольник, содержащий оба прямоугольника.
     **/
    public Rect union(Rect r) {
        return new Rect((this.x1 < r.x1) ? this.x1 : r.x1,
                        (this.y1 < r.y1) ? this.y1 : r.y1,
                        (this.x2 > r.x2) ? this.x2 : r.x2,
                        (this.y2 > r.y2) ? this.y2 : r.y2);
    }

    /**
     * Возвращается пересечение этого прямоугольника с другим.
     * Другими словами, возвращается область их наложения.
     **/
    public Rect intersection(Rect r) {
        Rect result =  new Rect((this.x1 > r.x1) ? this.x1 : r.x1,
                                (this.y1 > r.y1) ? this.y1 : r.y1,
                                (this.x2 < r.x2) ? this.x2 : r.x2,
                                (this.y2 < r.y2) ? this.y2 : r.y2);
        if (result.x1 > result.x2) { result.x1 = result.x2 = 0; }
        if (result.y1 > result.y2) { result.y1 = result.y2 = 0; }
        return result;
    }
    /**
      * Это метод нашего базового класса Object. Мы замещаем его так, чтобы
      * объекты класса Rect можно было осмысленно преобразовывать в строки,
      * складывать с другими строками при помощи оператора + и передавать
      * таким методам, как System.out.println().
      **/
    public String toString() {
        return "[" + x1 + "," + y1 + "; " + x2 + "," + y2 + "]";
    }
}
```

Тестирование класса Rect

Пример 2.2 – это самостоятельная программа `RectTest`, использующая по назначению класс `Rect` из примера 2.1. Обратите внимание на применение ключевого слова `new` и конструктора `Rect()` для создания нового объекта `Rect`. Программа использует оператор `.` (точка) для вызова методов объекта `Rect` и для доступа к его полям. Тестовая программа также неявно использует метод `toString()` класса `Rect`, когда при создании строк, отображаемых для пользователя, использует оператор конкатенации строк (+).

Пример 2.2. RectTest.java

```
package com.davidflanagan.examples.classes;
/** Этот класс демонстрирует возможные применения класса Rect */
public class RectTest {
    public static void main(String[] args) {
        Rect r1 = new Rect(1, 1, 4, 4);     // Создаем объекты класса Rect
        Rect r2 = new Rect(2, 3, 5, 6);
        Rect u = r1.union(r2);              // Вызываем методы класса Rect
        Rect i = r2.intersection(r1);

       if (u.isInside(r2.x1, r2.y1))   // Используем поля Rect и вызываем метод
            System.out.println("(" + r2.x1 + "," + r2.y1 +
                               ") содержится в объединении");

        // Следующие строки неявно вызывают метод Rect.toString()
        System.out.println(r1 + " union " + r2 + " = " + u);
        System.out.println(r1 + " intersect " + r2 + " = " + i);
    }
}
```

Подкласс класса Rect

Пример 2.3 – это простой подкласс класса `Rect` из примера 2.1. Этот класс `DrawableRect` наследует поля и методы класса `Rect` и добавляет собственный метод `draw()`, рисующий прямоугольник при помощи заданного объекта класса `java.awt.Graphics`. (Мы ближе познакомимся с классом `Graphics` в главе 11 «Графика».) `DrawableRect` определяет также конструктор, не делающий ничего, кроме того, что он передает свои аргументы соответствующему конструктору `Rect`. Обратите внимание на применение ключевого слова `extends` для обозначения того, что `Rect` является базовым классом класса `DrawableRect`.

Пример 2.3. DrawableRect.java

```
package com.davidflanagan.examples.classes;
/**
 * Это подкласс класса Rect, позволяющий нарисовать его на экране.
 * Он наследует все поля и методы класса Rect.
 * Для рисования он использует объект класса java.awt.Graphics.
```

```
 **/
public class DrawableRect extends Rect {
    /** Конструктор DrawableRect просто вызывает конструктор Rect()*/
    public DrawableRect(int x1, int y1, int x2, int y2) { super(x1,y1,x2,y2);
}

    /** Это новый метод, определяемый в классе DrawableRect */
    public void draw(java.awt.Graphics g) {
        g.drawRect(x1, y1, (x2 - x1), (y2-y1));
    }
}
```

Еще один подкласс

Пример 2.4 демонстрирует еще один подкласс. `ColoredRect` **является подклассом** `DrawableRect` **(см. пример 2.3) и, таким образом, подподклассом класса** `Rect` **(см. пример 2.1). Этот класс наследует поля и методы класса** `DrawableRect` **и класса** `Rect` **(и класса** `Object` **– неявного базового класса для** `Rect`**).** `ColoredRect` **вводит два новых поля, задающих цвет рамки и цвет заполнения изображаемого прямоугольника. (Эти поля принадлежат типу** `java.awt.Color`**, который мы изучим в главе 11.) Класс определяет также новый конструктор, позволяющий инициализировать эти поля. Наконец,** `ColoredRect` **замещает метод** `draw()` **класса** `DrawableRect`**. Метод** `draw()`**, определенный в классе** `ColoredRect`**, рисует прямоугольник с использованием цветов, заданных явно, а не по умолчанию, как это делал метод класса** `DrawableRect`**.**

Пример 2.4. ColoredRect.java

```
package com.davidflanagan.examples.classes;
import java.awt.*;
/**
 * Этот класс служит подклассом DrawableRect и добавляет цвета
 * изображаемого прямоугольника.
 **/
public class ColoredRect extends DrawableRect {
    // Это новые поля, определяемые этим классом.
    // Поля x1, y1, x2 и y2 наследуются от базового класса Rect.
    protected Color border, fill;

    /**
     * Этот конструктор использует super() для вызова конструктора
     * базового класса и сам тоже производит некую инициализацию.
     **/
    public ColoredRect(int x1, int y1, int x2, int y2,
                       Color border, Color fill)
    {
        super(x1, y1, x2, y2);
        this.border = border;
        this.fill = fill;
    }
```

```
        /**
         * Этот метод замещает метод draw() нашего базового класса так, чтобы
         * он мог воспользоваться заданными цветами.
         **/
        public void draw(Graphics g) {
            g.setColor(fill);
            g.fillRect(x1, y1, (x2-x1), (y2-y1));
            g.setColor(border);
            g.drawRect(x1, y1, (x2-x1), (y2-y1));
        }
    }
```

Комплексные числа

Пример 2.5 показывает определение класса, представляющего комплексные числа. Из уроков алгебры вы можете припомнить, что комплексное число – это сумма действительного числа и мнимого числа. Мнимое число i – это квадратный корень из -1. Класс `ComplexNumber` определяет два поля типа `double`, представляющих действительную и мнимую части числа. Эти поля объявлены как `private`. Это означает, что они могут использоваться только внутри тела класса; они недоступны извне класса. Поскольку эти поля недоступны, класс определяет два метода доступа, `real()` и `imaginary()`, просто возвращающие значения этих полей. Этот прием объявления полей как `private` и определения методов, осуществляющих доступ к ним, называется *инкапсуляцией* (*encapsulation*). Инкапсуляция скрывает от пользователя реализацию класса, в результате чего вы можете изменять реализацию, никак не задевая этим пользователей.

Заметьте, что класс `ComplexNumber` не определяет никаких методов, кроме конструктора, устанавливающего значения полей класса. Как только объект класса `ComplexNumber` создан, число, которое он представляет, уже не может быть изменено. Это свойство известно как *неизменяемость* (*immutability*); оно иногда бывает полезным при конструировании неизменяемых объектов.

`ComplexNumber` определяет два метода `add()` и два метода `multiply()`, осуществляющих сложение и умножение комплексных чисел. Различие между двумя версиями каждого из этих методов состоит в том, что один из них является методом экземпляра, а другой методом класса, или статическим методом. Рассмотрим для примера методы `add()`. Метод экземпляра прибавляет значение текущего экземпляра класса `ComplexNumber` к другому заданному объекту `ComplexNumber`. У метода класса нет текущего экземпляра; он просто складывает значения двух заданных объектов `ComplexNumber`. Метод экземпляра вызывается из экземпляра класса:

```
ComplexNumber sum = a.add(b);
```

Метод класса, напротив, вызывается из самого класса, а не из его экземпляра:

```
ComplexNumber sum = ComplexNumber.add(a, b);
```

Пример 2.5. ComplexNumber.java

```
package com.davidflanagan.examples.classes;
/**
 * Этот класс представляет комплексные числа и определяет методы, реализующие
 * арифметику комплексных чисел.
 **/
public class ComplexNumber {
  // Это переменные экземпляра. Каждый объект класса ComplexNumber содержит
  // два значения типа double, обозначаемые x и y. Они описаны как private и,
  // значит, недоступны извне этого класса. Доступ к ним осуществляется
  // посредством описанных ниже методов real() и imaginary().
  private double x, y;
  /** Это конструктор. Он инициализирует переменные x и y */
  public ComplexNumber(double real, double imaginary) {
    this.x = real;
    this.y = imaginary;
  }
  /**
   * Метод доступа. Возвращает действительную часть комплексного числа.
   * Обратите внимание на отсутствие метода setReal() для установки
   * действительной части числа. Это означает, что класс ComplexNumber
   * является «неизменяемым».
   **/
  public double real() { return x; }

  /** Метод доступа. Возвращает мнимую часть комплексного числа */
  public double imaginary() { return y; }

  /** Вычисляет абсолютную величину комплексного числа */
  public double magnitude() { return Math.sqrt(x*x + y*y); }
  /**
   * Этот метод преобразует ComplexNumber в строку. Это метод класса Object,
   * который мы замещаем так, чтобы комплексные числа можно было осмысленно
   * преобразовать в строки, которые удобно выводить посредством метода
   * System.out.println() и подобных ему
   **/
  public String toString() { return "{" + x + "," + y + "}"; }

  /**
   * Это статический метод класса. Он берет два комплексных числа, складывает
   * их и возвращает результат в виде третьего числа. Поскольку метод
   * статический, для него не существует «текущий экземпляр», или объект
   * «this». Применяется он так: ComplexNumber c = ComplexNumber.add(a, b);
   **/
  public static ComplexNumber add(ComplexNumber a, ComplexNumber b) {
    return new ComplexNumber(a.x + b.x, a.y + b.y);
  }
```

```
    /**
     * Это нестатический метод экземпляра с тем же именем. Он прибавляет
     * заданное комплексное число к текущему комплексному числу.
     * Применяется он так:
     * ComplexNumber c = a.add(b);
     **/
    public ComplexNumber add(ComplexNumber a) {
      return new ComplexNumber(this.x + a.x, this.y+a.y);
    }
    /** Статический метод класса для умножения комплексных чисел */
    public static ComplexNumber multiply(ComplexNumber a, ComplexNumber b) {
      return new ComplexNumber(a.x*b.x - a.y*b.y, a.x*b.y + a.y*b.x);
    }
    /** Метод экземпляра для умножения на комплексное число */
    public ComplexNumber multiply(ComplexNumber a) {
      return new ComplexNumber(x*a.x - y*a.y, x*a.y + y*a.x);
    }
}
```

Вычисление псевдослучайных чисел

До сих пор мы определяли классы, представляющие реальные объекты. (Комплексные числа – это абстрактное математическое понятие, но не так трудно вообразить их объектами, принадлежащими реальному миру.) В некоторых случаях, однако, требуется создать класс, не представляющий какого-либо рода объект или даже абстрактное понятие. В примере 2.6 определен класс, который может вычислять псевдослучайные числа и является именно таким классом.

Очевидно, что класс `Randomizer` не реализует никакого объекта. Однако оказывается, что простому алгоритму генерирования псевдослучайных чисел необходима переменная состояния (`seed`) для хранения первого элемента случайной последовательности. Раз вам понадобилось отслеживать какое-то состояние, вы уже не можете просто определить статический метод `random()`, как это делалось при определении методов `Factorial.factorial()` в главе 1. Если метод нуждается в некотором состоянии, сохраняемом между его последовательными вызовами, то его обычно следует сделать методом экземпляра объекта, содержащего это необходимое состояние, даже если у этого объекта нет очевидного двойника в реальном или абстрактном мире.

Таким образом, наш класс `Randomizer` определяет единственную переменную экземпляра, `seed`, в которой хранится состояние, необходимое для порождения последовательности псевдослучайных чисел. Другие поля в `Randomizer` объявлены как `static` и `final`, что в Java делает их константами. Другими словами, для каждого поля `static final` есть поле, ассоциированное с самим классом, значение которого никогда не меняется.

Пример 2.6. Randomizer.java

```
package com.davidflanagan.examples.classes;
/**
 * Этот класс определяет методы для вычисления псевдослучайных чисел
 * и определяет переменную состояния, которая должна сохраняться
 * для нужд этих методов.
 **/
public class Randomizer {
  // Тщательно подобранные константы из книги «Numerical Recipes in C».
  // Все поля, объявленные как «static final», являются константами.
  static final int m = 233280;
  static final int a = 9301;
  static final int c = 49297;
  // Переменная состояния, сохраняемая каждым экземпляром Randomizer
  int seed = 1;

  /**
   * Конструктор для класса Randomizer(). Ему нужно передать
   * произвольное начальное значение, «зерно» его псевдослучайности.
   **/
  public Randomizer(int seed) { this.seed = seed; }

  /**
   * Этот метод вычисляет псевдослучайные числа в промежутке между 0 и 1
   * с применением очень простого алгоритма. Методы Math.random()
   * и java.util.Random вычисляют случайные числа гораздо лучше.
   **/
  public float randomFloat() {
    seed = (seed * a + c) % m;
    return (float) Math.abs((float)seed/(float)m);
  }

  /**
   * Этот метод вычисляет псевдослучайные числа в промежутке между 0
   * и заданным максимальным значением. Он использует вышеприведенный
   * метод randomFloat().
   **/
  public int randomInt(int max) {
    return Math.round(max * randomFloat());
  }

  /**
   * Этот вложенный класс является простой тестовой программой: он печатает
   * 10 случайных целых чисел. Обратите внимание на то, как объект Randomizer
   * получает стартовое значение с применением текущего значения времени.
   **/
  public static class Test {
    public static void main(String[] args) {
      Randomizer r = new Randomizer((int)new java.util.Date().getTime());
      for(int i = 0; i < 10; i++) System.out.println(r.randomInt(100));
    }
  }
}
```

Пример 2.6 представляет новую важную возможность. Класс `Randomizer` определяет статический внутренний (inner) класс `Test`. Этот класс, `Randomizer.Test`, содержит метод `main()` и, таким образом, является самостоятельной программой, пригодной для тестирования класса `Randomizer`. В результате компилирования файла *Randomizer.java* вы получите два файла классов, *Randomizer.class* и *Randomizer$Test.class*. Запуск вложенного класса `Randomizer.Test` не совсем тривиален. Вы *имели бы право* написать:

```
% java com.davidflanagan.examples.classes.Randomizer.Test
```

Тем не менее действующая версия Java SDK не умеет корректно связать имя класса `Randomizer.Test` с файлом класса *Randomizer$Test.class*. Поэтому, чтобы запустить тестовую программу, необходимо вызывать интерпретатор Java, используя символ `$` вместо символа «.» в имени класса:

```
% java com.davidflanagan.examples.classes.Randomizer$Test
```

Однако следует иметь в виду, что в Unix-системах символ `$` имеет специальный смысл и при вводе должен предваряться обратным слэшем. Следовательно, в Unix-системах следует вводить:

```
% java com.davidflanagan.examples.classes.Randomizer\$Test
```

или:

```
% java 'com.davidflanagan.examples.classes.Randomizer$Test'
```

Этот прием необходим для запуска программы на Java, определенной как внутренний класс.

Статистические вычисления

Пример 2.7 демонстрирует класс, вычисляющий простые статистические функции, работающие с набором чисел. Как только числа переданы методу `addDatum()`, класс `Averager` обновляет свое внутреннее состояние, так что другие методы этого класса могут с легкостью возвратить среднее и стандартное отклонение переданного к этому моменту набора чисел. Подобно классу `Randomizer`, класс `Averager` не представляет никакого реального объекта или абстрактного понятия. Тем не менее, `Averager` сохраняет некое состояние (на этот раз в виде полей `private`) и содержит методы, оперирующие этим состоянием, то есть он реализован как класс.

Подобно примеру 2.6, пример 2.7 определяет внутренний класс `Test`, который содержит метод `main()`, реализующий тестовую программу для класса `Averager`.

Пример 2.7. Averager.java

```
package com.davidflanagan.examples.classes;
/**
 * Класс, вычисляющий текущее среднее переданных ему чисел
 **/
public class Averager {
  // Поля Private для хранения текущего состояния.
  private int n = 0;
  private double sum = 0.0, sumOfSquares = 0.0;
  /**
   * Этот метод учитывает новое введенное значение.
   **/
  public void addDatum(double x) {
    n++;
    sum += x;
    sumOfSquares += x * x;
  }

  /** Этот метод возвращает среднее всех чисел, переданных addDatum() */
  public double getAverage() { return sum / n; }
  /** Этот метод возвращает стандартное отклонение */
  public double getStandardDeviation() {
     return Math.sqrt(((sumOfSquares - sum*sum/n)/n));
  }
  /** Этот метод возвращает количество чисел, переданных addDatum() */
  public double getNum() { return n; }
  /** Этот метод возвращает сумму всех чисел, переданных addDatum() */
  public double getSum() { return sum; }
  /** Этот метод возвращает сумму квадратов всех чисел, переданных addDatum() */
  public double getSumOfSquares() { return sumOfSquares; }
  /** Этот метод «сбрасывает» объект Averager, чтобы начать считать
      «с чистого листа» */
  public void reset() { n = 0; sum = 0.0; sumOfSquares = 0.0; }
  /**
   * Этот вложенный класс является простой тестовой программой, помогающей
   * убедиться, что наши коды работают правильно.
   **/
  public static class Test {
    public static void main(String args[]) {
      Averager a = new Averager();
      for(int i = 1; i <= 100; i++) a.addDatum(i);
      System.out.println("Среднее: " + a.getAverage());
      System.out.println("Стандартное отклонение: " +
                 a.getStandardDeviation());
      System.out.println("N: " + a.getNum());
      System.out.println("Сумма: " + a.getSum());
      System.out.println("Сумма квадратов: " + a.getSumOfSquares());
    }
  }
}
```

Класс связанных списков

Пример 2.8 предъявляет класс, LinkedList, реализующий структуру связанного списка. Пример определяет также интерфейс Linkable. Если объект должен быть связан с LinkedList, класс этого объекта должен реализовывать интерфейс Linkable. Напомню, что интерфейс определяет методы, но не обеспечивает эти методы конкретным содержанием. Класс реализует интерфейс, обеспечивая реализацию каждого метода в интерфейсе и применяя ключевое слово implements в своем объявлении. Любой экземпляр класса, реализующий интерфейс Linkable, можно рассматривать как экземпляр Linkable. Объект класса LinkedList рассматривает все объекты, входящие в его список, как экземпляры интерфейса Linkable и, следовательно, не обязан ничего знать об их действительном типе.

Обратите внимание, что этот пример был написан для Java 1.1. Java 1.2 вводит подобный, но независимый класс java.util.LinkedList. Этот новый класс коллекций LinkedList полезнее класса, разработанного в примере 2.8, но пример 2.8 – лучший пример применения интерфейсов.

LinkedList определяет одно поле состояния (head), указывающее на первый элемент Linkable в списке. Класс определяет также множество методов для добавления и удаления элементов списка. Обратите внимание, что интерфейс Linkable вложен в класс LinkedList. Хотя это удобный и полезный способ определения интерфейса, нет никакой необходимости вкладывать его таким образом. Linkable можно было легко определить как обычный интерфейс верхнего уровня.

Класс LinkedList определяет также внутренний класс Test, как и в предыдущих примерах, являющийся самостоятельной программой для тестирования класса. В этом примере, однако, внутренний класс Test сам содержит внутренний класс LinkableInteger. Этот класс реализует Linkable; его экземпляры присоединяет к списку тестовая программа.

Пример 2.8. LinkedList.java

```
package com.davidflanagan.examples.classes;
/**
 * Этот класс реализует связанный список объектов, способный хранить объекты
 * любого типа, реализующие вложенный интерфейс Linkable. Обратите внимание,
 * что все методы синхронизированы, так что класс могут безопасно
 * использовать несколько потоков одновременно.
 **/
public class LinkedList {
  /**
   * Этот интерфейс определяет методы, необходимые любому объекту
   * для присоединения к связанному списку.
   **/
  public interface Linkable {
    public Linkable getNext();    // Возвращает следующий элемент списка
    public void setNext(Linkable node); // Устанавливает следующий элемент списка
```

```
}
// У этого класса есть принимаемый по умолчанию конструктор:
// public LinkedList() {}
/** Это единственное поле класса. Оно содержит первый узел (head) списка*/
Linkable head;
/** Возвращает первый узел списка */
public synchronized Linkable getHead() { return head; }
/** Вставляет узел в начало списка */
public synchronized void insertAtHead(Linkable node) {
  node.setNext(head);
  head = node;
}
/** Вставляет узел в конец списка */
public synchronized void insertAtTail(Linkable node) {
  if (head == null) head = node;
  else {
    Linkable p, q;
    for(p = head; (q = p.getNext()) != null; p = q) /* пустой цикл */;
    p.setNext(node);
  }
}
/** Удаляет и возвращает в качестве значения узел,
    находящийся в начале списка */
public synchronized Linkable removeFromHead() {
  Linkable node = head;
  if (node != null) {
    head = node.getNext();
    node.setNext(null);
  }
  return node;
}
/** Удаляет и возвращает в качестве значения узел,
    находящийся в конце списка */
public synchronized Linkable removeFromTail() {
  if (head == null) return null;
  Linkable p = head, q = null, next = head.getNext();
  if (next == null) {
    head = null;
    return p;
  }
  while((next = p.getNext()) != null) {
    q = p;
    p = next;
  }
  q.setNext(null);
  return p;
}
/**
 * Удаляет из списка узел, совпадающий с заданным. Для проверки
 * на совпадение вместо оператора == применяется метод equals().
 **/
```

```
public synchronized void remove(Linkable node) {
  if (head == null) return;
  if (node.equals(head)) {
    head = head.getNext();
    return;
  }
  Linkable p = head, q = null;
  while((q = p.getNext()) != null) {
    if (node.equals(q)) {
      p.setNext(q.getNext());
      return;
    }
    p = q;
  }
}

/** Этот вложенный класс определяет метод main(), тестирующий LinkedList */
public static class Test {
  /**
   * Это тестовый класс, реализующий интерфейс Linkable
   **/
  static class LinkableInteger implements Linkable {
    int i;     // Данные, содержащиеся в узле
    Linkable next; // Ссылка на следующий узел в списке
    public LinkableInteger(int i) { this.i = i; }        // Конструктор
    public Linkable getNext() { return next; }           // Часть Linkable
    public void setNext(Linkable node) { next = node; }  // Linkable
    public String toString() { return i + ""; }  // Для удобства печати
    public boolean equals(Object o) {            // Для сравнения
      if (this == o) return true;
      if (!(o instanceof LinkableInteger)) return false;
      if (((LinkableInteger)o).i == this.i) return true;
      return false;
    }
  }
  /**
   * Тестовая программа. Вставляет несколько узлов, удаляет несколько
   * узлов, затем печатает все элементы списка. Она должна напечатать
   * числа 4, 6, 3, 1 и 5
   **/
  public static void main(String[] args) {
    LinkedList ll = new LinkedList();         // Создание списка
    ll.insertAtHead(new LinkableInteger(1));  // Что-то вставляется
    ll.insertAtHead(new LinkableInteger(2));
    ll.insertAtHead(new LinkableInteger(3));
    ll.insertAtHead(new LinkableInteger(4));
    ll.insertAtTail(new LinkableInteger(5));
    ll.insertAtTail(new LinkableInteger(6));
    System.out.println(ll.removeFromHead()); // Удаляем и печатаем узел
    System.out.println(ll.removeFromTail()); // Снова удаляем и печатаем
    ll.remove(new LinkableInteger(2));       // Удаляем еще один узел
```

```
        // Теперь печатаем содержимое списка.
        for(Linkable l = ll.getHead(); l != null; l = l.getNext())
          System.out.println(l);
      }
    }
  }
```

Усовершенствованная сортировка

В главе 1 мы видели пример простого, незатейливого алгоритма сортировки массива чисел. Пример 2.9 определяет класс `Sorter`, который поддерживает более эффективную и универсальную сортировку. `Sorter` определяет множество статических методов `sort()`, каждый из которых принимает несколько иные аргументы, чем другие методы. Некоторые из этих методов разными способами сортируют строки, остальные сортируют другие объекты. Последний метод `sort()` реализует алгоритм быстрой сортировки (quicksort) для эффективной сортировки массива объектов. Все другие методы – это варианты, каждый из которых, в конечном счете, вызывает общий метод сортировки.

Методы `sort()`, сортирующие строки, пользуются появившимися в Java 1.1 возможностями, связанными с интернационализацией. В частности, они используют классы `java.util.Locale`, `java.text.Collator` и `java.text.CollationKey`. Подробнее мы изучим эти классы в главе 7 «Интернационализация».

Для сортировки массива объектов классу `Sorter` нужен какой-либо способ сравнения объектов, чтобы определить, в каком порядке они должны находиться в отсортированном массиве. `Sorter` определяет два вложенных интерфейса, `Sorter.Comparer` и `Sorter.Comparable`, предоставляющие различные способы реализации этого сравнения. Вы можете сортировать произвольные объекты, передавая объект `Comparer` одному из методов `sort()`. Объект `Comparer` определяет метод `compare()`, сравнивающий произвольные объекты. В качестве альтернативы классы сортируемых объектов могут реализовать интерфейс `Sorter.Comparable`. В этом случае сами объекты будут иметь методы `compareTo()`, и их можно будет сравнивать непосредственно.

Обратите внимание на то, что в Java 1.2 и в более поздних версиях Java класс `java.util.Arrays` определяет множество методов `sort()` для сортировки массивов объектов или примитивных значений. Также `java.util.Collections` определяет методы `sort()` для сортировки объектов `java.util.List` (эти классы входят в набор коллекций Java, появившийся в Java 1.2). Эти классы и методы предпочтительнее разработанных здесь сортирующих методов. Тем не менее пример 2.9 остается интересным и полезным. Обратите внимание также на то, что сортирующие методы в `Arrays` и `Collections` используют интерфейсы `java.util.Comparator` и `java.lang.Comparable`, подобные интерфейсам `Comparer` и `Comparable` из этого примера.

Пример 2.9 завершает эту главу. Если вы уже забежали вперед и взглянули на программу, то, вероятно, заметили, что это довольно сложный пример. Будучи таким, он заслуживает тщательного изучения. В частности, в этой программе интенсивно используются внутренние классы, так что перед подробным разбором кода следует хорошо понимать, как работают внутренние классы. Как обычно в конце примера находится внутренний класс `Test`, но внутренние классы и интерфейсы используются на протяжении всей программы.

Пример 2.9. Sorter.java

```
package com.davidflanagan.examples.classes;
// Это классы, которые понадобятся для сортировки многоязыковых строк
import java.text.Collator;
import java.text.CollationKey;
import java.util.Locale;
/**
 * Этот класс определяет набор статических методов для эффективной сортировки
 * массивов строк или других объектов. Он определяет также два интерфейса,
 * обеспечивающие два различных способа сравнения сортируемых объектов.
 **/
public class Sorter {
  /**
   * Этот интерфейс определяет метод compare(), используемый для сравнения
   * двух объектов. Для сортировки объектов некоторого типа вы должны
   * снабдить объект Comparer методом compare(), упорядочивающим эти объекты
   * желательным для вас образом.
   **/
  public static interface Comparer {
    /**
     * Объекты сравниваются, и возвращается значение, обозначающее
     * их относительный порядок:
     * если (a > b), возвращается значение > 0;
     * если (a == b), возвращается значение 0;
     * если (a < b), возвращается значение < 0.
     **/
    public int compare(Object a, Object b);
  }
  /**
   * Это альтернативный интерфейс, который можно использовать
   * для упорядочивания объектов. Если класс реализует интерфейс Comparable,
   * то любые два экземпляра этого класса можно сравнить непосредственно
   * путем вызова метода compareTo().
   **/
  public static interface Comparable {
    /**
     * Объекты сравниваются, и возвращается значение, обозначающее
     * их относительный порядок:
     * если (a > b), возвращается значение > 0;
     * если (a == b), возвращается значение 0;
     * если (a < b), возвращается значение < 0.
```

```
   **/
  public int compareTo(Object other);
}
/**
 * Это внутренний объект Comparer (созданный безымянным классом),
 * сравнивающий две строки ASCII. Он используется ниже методами sortAscii.
 **/
private static Comparer ascii_comparer = new Comparer() {
    public int compare(Object a, Object b) {
      return ((String)a).compareTo((String)b);
    }
  };

/**
 * Это еще один внутренний объект Comparer. Он используется для сравнения
 * двух объектов Comparable. Ниже он используется методами sort(),
 * принимающими в качестве аргументов объекты Comparable вместо
 * произвольных объектов.
 **/
private static Comparer comparable_comparer = new Comparer() {
    public int compare(Object a, Object b) {
      return ((Comparable)a).compareTo(b);
    }
  };

/** Массив строк ASCII сортируется по возрастанию */
public static void sortAscii(String[] a) {
  // Обратите внимание на применение объекта ascii_comparer
  sort(a, null, 0, a.length-1, true, ascii_comparer);
}

/**
 * Часть массива строк ASCII сортируется по возрастанию или по убыванию
 * в зависимости от аргумента up.
 **/
public static void sortAscii(String[] a, int from, int to, boolean up) {
  // Обратите внимание на применение объекта ascii_comparer
  sort(a, null, from, to, up, ascii_comparer);
}

/** Массив строк ASCII сортируется по возрастанию без учета регистра */
public static void sortAsciiIgnoreCase(String[] a) {
  sortAsciiIgnoreCase(a, 0, a.length-1, true);
}

/**
 * Часть массива строк ASCII сортируется без учета регистра: по возрастанию,
 * если up имеет значение true, в противном случае – по убыванию.
 **/
public static void sortAsciiIgnoreCase(String[] a, int from, int to,
                    boolean up) {
  if ((a == null) || (a.length < 2)) return;
  // Создается вторичный массив строк, содержащий строчную
  // (в нижнем регистре) версию всех заданных строк.
```

```
  String b[] = new String[a.length];
  for(int i = 0; i < a.length; i++) b[i] = a[i].toLowerCase();
  // Сортируется этот вторичный массив, и изначальный массив
  // переупорядочивается точно в таком же порядке, в результате получается
  // сортировка, не зависящая от регистра. Обратите внимание
  // на применение объекта ascii_comparer
  sort(b, a, from, to, up, ascii_comparer);
}

/**
 * Массив строк сортируется по возрастанию с применением
 * принятого по умолчанию местного порядка символов
 **/
public static void sort(String[] a) {
  sort(a, 0, a.length-1, true, false, null);
}

/**
 * Часть массива строк сортируется с применением принятого по умолчанию
 * местного порядка символов: по возрастанию - если аргумент up имеет
 * значение true, в противном случае - по убыванию. Если ignorecase имеет
 * значение true, различие между строчными и заглавными буквами игнорируется.
 **/
public static void sort(String[] a, int from, int to,
            boolean up, boolean ignorecase) {
  sort(a, from, to, up, ignorecase, null);
}

/**
 * Часть массива строк сортируется с применением принятого по умолчанию
 * местного порядка символов: по возрастанию - если аргумент up имеет
 * значение true, в противном случае - по убыванию. Если ignorecase имеет
 * значение true, различие между строчными и заглавными буквами игнорируется.
 **/
public static void sort(String[] a, int from, int to,
            boolean up, boolean ignorecase,
            Locale locale) {
  // Не сортируем, если сортировать нечего
  if ((a == null) || (a.length < 2)) return;

  // Объект java.text.Collator осуществляет интернационализованное
  // сравнение строк. Он создается для заданной или принимаемой
  // по умолчанию локализации.
  Collator c;
  if (locale == null) c = Collator.getInstance();
  else c = Collator.getInstance(locale);

  // Указывается, следует ли при сортировке принимать во внимание регистр.
  // Замечание: этот вариант, кажется, не будет корректно работать в JDK 1.1.1
  // при использовании принимаемой по умолчанию локализации American English.
  if (ignorecase) c.setStrength(Collator.secondary);

  // Применяется объект Collator для создания массива объектов
  // CollationKey, соответствующих каждой из строк. Сравнение объектов
```

```
  // CollationKeys происходит значительно быстрее сравнения строк.
  CollationKey[] b = new CollationKey[a.length];
  for(int i = 0; i < a.length; i++) b[i] = c.getCollationKey(a[i]);
  // Теперь определяется объект Comparer для сравнения объектов
  // CollationKey с применением безымянного класса.
  Comparer comp = new Comparer() {
      public int compare(Object a, Object b) {
        return ((CollationKey)a).compareTo((CollationKey)b);
      }
    };

  // Наконец, сортируется массив объектов CollationKey,
  // и изначальный массив строк переупорядочивается точно так же.
  sort(b, a, from, to, up, comp);
}

/** Массив объектов Comparable сортируется по возрастанию */
public static void sort(Comparable[] a) {
  sort(a, null, 0, a.length-1, true);
}

/**
 * Сортируется часть массива объектов Comparable: по возрастанию -
 * если аргумент up имеет значение true, в противном случае - по убыванию.
 **/
public static void sort(Comparable[] a, int from, int to, boolean up) {
  sort(a, null, from, to, up, comparable_comparer);
}

/**
 * Сортируется часть массива объектов Comparable a: по возрастанию -
 * если аргумент up имеет значение true, в противном случае - по убыванию.
 * Массив b переупорядочивается точно в том же порядке, что и массив a.
 **/
public static void sort(Comparable[] a, Object[] b,
            int from, int to, boolean up) {
  sort(a, b, from, to, up, comparable_comparer);
}

/**
 * Массив произвольных объектов сортируется по возрастанию
 * с применением сравнения, определенного объектом Comparer c
 **/
public static void sort(Object[] a, Comparer c) {
  sort(a, null, 0, a.length-1, true, c);
}

/**
 * Часть массива объектов сортируется с применением сравнения,
 * определенного объектом Comparer c: по возрастанию - если аргумент up
 * имеет значение true, в противном случае - по убыванию.
 **/
public static void sort(Object[] a, int from, int to, boolean up,
            Comparer c)
```

```
{
  sort(a, null, from, to, up, c);
}

/**
 * Это главная процедура sort(). Она выполняет быструю сортировку (quicksort)
 * массива a от элемента from до элемента to. Аргумент up
 * указывает, следует ли сортировать в возрастающем (true) или
 * убывающем (false) порядке. Аргумент Comparer «c» используется
 * для сравнения элементов массива. Элементы массива b
 * переупорядочиваются точно так же, как элементы массива a.
 **/
public static void sort(Object[] a, Object[] b,
            int from, int to,
            boolean up, Comparer c)
{
  // If there is nothing to sort, return
  if ((a == null) || (a.length < 2)) return;

  // Это основной алгоритм быстрой сортировки, из которого убраны
  // «украшения», делающие его еще быстрее, но и еще запутаннее,
  // чем сейчас. Вы должны понимать, что делает программа, не выясняя,
  // где гарантия того, что массив будет отсортирован...
  // Обратите внимание на применение метода compare() объекта Comparer.
  int i = from, j = to;
  Object center = a[(from + to) / 2];
  do {
    if (up) { // по возрастанию
      while((i < to) && (c.compare(center, a[i]) > 0)) i++;
      while((j > from) && (c.compare(center, a[j]) < 0)) j--;
    } else {  // по убыванию
      while((i < to) && (c.compare(center, a[i]) < 0)) i++;
      while((j > from) && (c.compare(center, a[j]) > 0)) j--;
    }
    if (i < j) {
     Object tmp = a[i]; a[i] = a[j]; a[j] = tmp; // переставляются элементы
     if (b != null) { tmp = b[i]; b[i] = b[j]; b[j] = tmp; } //  перестановка
    }
    if (i <= j) { i++; j--; }
  } while(i <= j);
  if (from < j) sort(a, b, from, j, up, c);      // рекурсивно сортируется
                                                 // оставшееся
  if (i < to) sort(a, b, i, to, up, c);
}
/**
 * Этот вложенный класс определяет тестовую программу, демонстрирующую
 * несколько способов применения класса Sorter для сортировки
 * объектов класса ComplexNumber
 **/
public static class Test {
  /**
   * Этот подкласс класса ComplexNumber реализует интерфейс Comparable
   * и определяет метод compareTo() для сравнения комплексных чисел.
```

```
 * Он сравнивает числа по их абсолютной величине. То есть по их расстоянию
 * от начала координат.
 **/
static class SortableComplexNumber extends ComplexNumber
  implements Sorter.Comparable {
  public SortableComplexNumber(double x, double y) { super(x, y); }
  public int compareTo(Object other) {
    return sign(this.magnitude()-((ComplexNumber)other).magnitude());
  }
}

/** Тестовая программа, сортирующая комплексные числа разными способами. */
public static void main(String[] args) {
  // Определяется массив объектов SortableComplexNumber.
  // Он инициализируется случайными комплексными числами.
  SortableComplexNumber[] a = new SortableComplexNumber[5];
  for(int i = 0; i < a.length; i++)
    a[i] = new SortableComplexNumber(Math.random()*10,
                    Math.random()*10);

  // Теперь осуществляется сортировка с применением метода compareTo()
  // класса SortableComplexNumber, упорядочивающего числа
  // по абсолютной величине. Результат печатается.
  System.out.println("Отсортированы по абсолютной величине:");
  Sorter.sort(a);
  for(int i = 0; i < a.length; i++) System.out.println(a[i]);

  // Снова сортируются комплексные числа с применением объекта Comparer,
  // сравнивающего числа на основании сумм их действительной
  // и мнимой частей.
  System.out.println("Отсортированы по сумме действительной
                      и мнимой частей:");
  Sorter.sort(a, new Sorter.Comparer() {
      public int compare(Object a, Object b) {
        ComplexNumber i = (ComplexNumber)a;
        ComplexNumber j = (ComplexNumber)b;
        return sign((i.real() + i.imaginary()) -
              (j.real() + j.imaginary()));
      }
    });
  for(int i = 0; i < a.length; i++) System.out.println(a[i]);
  // Они сортируются снова с применением объекта Comparer, сравнивающего
  // сначала их действительные, а затем мнимые части.
  System.out.println("Отсортированы в порядке убывания сначала
                      по действительной, а затем по мнимой частям:");
  Sorter.sort(a, 0, a.length-1, false, new Sorter.Comparer() {
      public int compare(Object a, Object b) {
        ComplexNumber i = (ComplexNumber) a;
        ComplexNumber j = (ComplexNumber) b;
        double result = i.real() - j.real();
        if (result == 0) result = i.imaginary()-j.imaginary();
        return sign(result);
```

```
        }
      });
    for(int i = 0; i < a.length; i++) System.out.println(a[i]);
  }
  /** Это вспомогательная процедура, используемая программами сравнения */
  public static int sign(double x) {
    if (x > 0) return 1;
    else if (x < 0) return -1;
    else return 0;
  }
 }
}
```

Упражнения

2-1. Напишите класс `Circle`, подобный классу `Rect`. Определите методы `move()` и `isInside()`. (Напомню, что окружность определяется как множество точек, находящихся на заданном расстоянии от центра, называемом *радиусом окружности*. Проверяйте принадлежность точки кругу, применяя теорему Пифагора для вычисления расстояния от точки до центра окружности.) Определите также метод `boundingBox()`, возвращающий наименьший объект `Rect`, целиком содержащий объект `Circle`. Напишите простую программу, тестирующую реализованные вами методы.

2-2. Напишите класс, представляющий почтовый адрес человека. Он должен иметь отдельные поля для указания имени, адреса (улица, номер дома, номер квартиры), города, штата и ZIP-кода. Определите метод `toString()`, который красиво форматирует вывод.

2-3. Применяя интерфейсы `Sort.Comparer` и/или `Sort.Comparable`, напишите статический метод `search()` для класса `Search`, осуществляющий эффективный двоичный поиск заданного объекта в отсортированном массиве объектов. Если объект найден в массиве, `search()` должен возвращать индекс объекта в массиве, в противном случае он должен возвращать –1.

Глава 3

Ввод/вывод

От компьютерной программы не слишком много пользы, если она не умеет общаться с внешним миром. Такое общение часто происходит в форме ввода/вывода (input/output, I/O). Возможности средств ввода/вывода – основная характеристика любой платформы программирования. В Java ввод и вывод осуществляются при помощи классов и интерфейсов пакета `java.io`. В данной главе демонстрируются многие средства ввода/вывода из этого пакета. Примеры показывают, как:

- осуществлять чтение/запись файлов;
- получать список каталогов и информацию о размерах и датировании файлов;
- применять различные классы потоков Java;
- определять пользовательские подклассы потоков.

Приемы, представленные здесь, применяются и в других главах книги. Множество примеров, использующих для ввода/вывода потоки, рассмотрены в главе 5 «Сетевые операции», а в главе 9 «Сериализация объектов» мы познакомимся с вводом/выводом специального вида.

Файлы и потоки

Одним из часто применяемых классов пакета `java.io` является `File`. Название этого класса может ввести в заблуждение, так как он представляет скорее имя файла (или название каталога), нежели сам файл. Поскольку файлы (и каталоги) в разных операционных системах подчиняются разным соглашениям об именах, Java предлагает класс `File`, пытающий скрыть некоторые из этих различий. Класс `File` определяет также различные методы для операций над файлами в целом: удале-

ния файлов, создания каталогов, вывода списка каталогов, запроса размера и времени изменения файла и т. д.

Класс `File`, обеспечивая нас методами для оперирования каталогами и файлами, содержащимися в этих каталогах, не предоставляет никаких методов, манипулирующих содержимым файлов. Другими словами, он не предоставляет никаких способов чтения или записи байтов или символов, содержащихся в файлах. В Java последовательный файловый ввод/вывод осуществляется в рамках потоков. (Файловый ввод/вывод произвольного доступа осуществляется при помощи класса `RandomAccessFile`, но последовательный ввод/вывод – гораздо более общий случай.)

Поток (*stream*) – это просто объект, из которого/в который данные могут последовательно считываться/записываться. Основной объем пакета `java.io` занимают 40 классов потоков. Классы `InputStream` и `OutputStream` и их соответствующие подклассы являются объектами, предназначенными для чтения и записи байтовых потоков, тогда как классы `Reader` и `Writer` и их подклассы являются объектами для чтения и записи потоков символов Unicode. Помимо этих классов потоков пакет `java.util.zip` определяет 8 байтовых потоков ввода и вывода для сжатия и восстановления данных. В таблицах с 3.1 по 3.4 представлен обзор классов потоков, содержащихся в пакетах `java.io` и `java.util.zip`.

Таблица 3.1. Байтовые потоки ввода

Байтовые потоки ввода	Описание
`BufferedInputStream`	Считывает байты из класса `InputStream` в буфер и затем возвращает байты из буфера, что повышает эффективность считывания небольших объемов данных
`ByteArrayInputStream`	Последовательно считывает байты из массива
`CheckedInputStream`	Этот класс пакета `java.util.zip` вычисляет контрольную сумму байтов, считанных им из класса `InputStream`
`DataInputStream`	Считывает двоичные представления примитивных типов Java из класса `InputStream`
`FileInputStream`	Последовательно считывает байты из файла
`FilterInputStream`	Базовый класс для классов, фильтрующих байтовые потоки ввода
`GZIPInputStream`	Этот класс пакета `java.util.zip` восстанавливает сжатые при помощи алгоритма GZIP байты, считанные этим же классом из класса `InputStream`
`InflaterInputStream`	Базовый класс для классов `GZIPInputStream` и `ZipInputStream`
`InputStream`	Базовый класс всех байтовых потоков ввода
`LineNumberInputStream`	Применение этого класса – как принадлежащего Java 1.1 – не рекомендуется; используйте вместо него класс `LineNumberReader`

Байтовые потоки ввода	Описание
ObjectInputStream	Считывает из байтового потока двоичные представления объектов и примитивных значений Java. Этот класс применяется для десериализации объектов
PipedInputStream	Считывает байты из потока PipedOutputStream, к которому он присоединен. Применяется в программах с несколькими параллельными потоками исполнения (многопоточных программах, multithreaded program)
PushbackInputStream	Добавляет к потоку ввода буфер фиксированного размера с «выталкиванием назад» (pushback), обеспечивающий отмену считывания байтов. Полезен при работе с некоторыми анализаторами (parser)
SequenceInputStream	Последовательно считывает байты из двух или из большего числа потоков ввода, как если бы они были одним потоком
StringBufferInputStream	Применение этого класса – как принадлежащего Java 1.1 – не рекомендуется; используйте вместо него класс StringReader
ZipInputStream	Этот класс пакета java.util.zip распаковывает входной ZIP-файл

Таблица 3.2. Символьные потоки ввода

Символьные потоки ввода	Описание
BufferedReader	Считывает байты из класса Reader в буфер и затем возвращает байты из буфера, что повышает эффективность считывания небольших объемов данных
CharArrayReader	Последовательно считывает символы из массива
FileReader	Последовательно считывает символы из файла. Является подклассом класса InputStreamReader, считывающего из автоматически создаваемого класса FileInputStream
FilterReader	Базовый класс для классов, фильтрующих символьные потоки ввода
InputStreamReader	Считывает символы из байтового потока ввода. Преобразует байты в символы в соответствии с принимаемой по умолчанию местной или явно заданной кодировкой
LineNumberReader	Считывает строки текста, отслеживая число прочитанных строк
PipedReader	Считывает символы, записанные в класс PipedWriter, к которому он присоединен. Применяется в многопоточных программах

Таблица 3.2 (продолжение)

PushbackReader	Добавляет к Reader буфер фиксированного размера с «выталкиванием назад», обеспечивающий отмену считывания символов. Полезен при работе с некоторыми анализаторами
Reader	Базовый класс всех символьных потоков ввода
StringReader	Последовательно считывает символы из строки

Таблица 3.3. Байтовые потоки вывода

Байтовые потоки вывода	Описание
BufferedOutputStream	С целью повышения эффективности буферизует вывод байтов; пишет в класс OutputStream, только когда буфер заполняется
ByteArrayOutputStream	Последовательно записывает байты в массив
CheckedOutputStream	Этот класс пакета java.util.zip вычисляет контрольную сумму байтов, записанных им в класс OutputStream
DataOutputStream	Записывает двоичные представления примитивных типов Java в класс OutputStream
DeflaterOutputStream	Базовый класс для классов GZIPOutputStream и ZipOutputStream
FileOutputStream	Последовательно записывает байты в файл
FilterOutputStream	Базовый класс для классов, фильтрующих байтовые потоки вывода
GZIPOutputStream	Этот класс пакета java.util.zip организует вывод введенных в него байтов со сжатием по алгоритму GZIP
ObjectOutputStream	Записывает в класс OutputStream двоичные представления объектов и примитивных значений Java. Этот класс применяется для сериализации объектов
OutputStream	Базовый класс всех байтовых потоков вывода
PipedOutputStream	Записывает байты в класс PipedInputStream, к которому он присоединен. Применяется в многопоточных программах
PrintStream	Записывает текстовые представления объектов и примитивных значений Java. Применение этого класса – как принадлежащего Java 1.1 – не рекомендуется за исключением его использования стандартным потоком вывода System.out. В других контекстах используйте вместо него класс PrintWriter
ZipOutputStream	Этот класс пакета java.util.zip сжимает записи в ZIP-файл

Таблица 3.4. Символьные потоки вывода

Символьные потоки вывода	Описание
BufferedWriter	С целью повышения эффективности буферизует вывод байтов; пишет в класс Writer, только когда буфер заполняется
CharArrayWriter	Последовательно записывает символы в массив
FileWriter	Последовательно записывает символы в файл. Является подклассом класса OutputStreamWriter, который автоматически создает класс FileOutputStream
FilterWriter	Базовый класс для классов, фильтрующих символьные потоки вывода
OutputStreamWriter	Записывает символы в байтовый поток вывода. Преобразует символы в байты в соответствии с принимаемой по умолчанию местной или заданной явно кодировкой
PipedWriter	Записывает символы в класс PipedReader, к которому он присоединен. Применяется в многопоточных программах
PrintWriter	Записывает текстовые представления объектов и примитивных значений Java в класс Writer
StringWriter	Последовательно записывает символы в создаваемый им же класс StringBuffer
Writer	Базовый класс всех символьных потоков вывода

Работа с файлами

Пример 3.1 представляет собой относительно короткую программу, удаляющую файл или каталог, заданный в командной строке. Он демонстрирует множество методов класса File – методов, оперирующих файлом (или каталогом) как единым целым, но не его содержимым. К числу других полезных методов класса File относятся getParent(), length(), mkdir() и renameTo().

Пример 3.1. Delete.java

```
package com.davidflanagan.examples.io;
import java.io.*;

/**
 * Этот класс состоит из статического метода delete() и самостоятельной
 * программы, удаляющей заданный файл или каталог.
 **/
public class Delete {
    /**
     * Это метод main() самостоятельной программы. После проверки своих
```

```
     * аргументов он вызывает метод Delete.delete() для выполнения удаления
     **/
    public static void main(String[] args) {
        if (args.length != 1) {    // Проверяем аргументы, передаваемые
                                   // в командной строке.
           System.err.println("Формат команды: java Delete <файл или каталог>");
            System.exit(0);
        }
        // Вызывается метод delete() и отображаются выданные
        // им сообщения об ошибках.
        try { delete(args[0]); }
        catch (IllegalArgumentException e) {
            System.err.println(e.getMessage());
        }
    }

    /**
     * Статический метод, выполняющий удаление. Вызывается методом main(), но
     * может вызываться и другими программами. Первым делом, прежде чем
     * попытаться их удалить, он убеждается, что заданные файл или каталог
     * допускают удаление. Встретив затруднение, он генерирует исключение
     * IllegalArgumentException.
     **/
    public static void delete(String filename) {
        // Создаем объект класса File для представления имени файла
        File f = new File(filename);

        // Убеждаемся, что файл или каталог существует и не защищен от записи
        if (!f.exists()) fail("Delete: нет такого файла или каталога:
                              " +filename);
        if (!f.canWrite()) fail("Delete: защищен от записи: " + filename);

        // Если это каталог, убеждаемся, что он пуст
        if (f.isDirectory()) {
            String[] files = f.list();
            if (files.length > 0)
                fail("Delete: каталог не пустой: " + filename);
        }

        // Если все проверки пройдены, пытаемся удалить
        boolean success = f.delete();

       // И генерируем исключение, если по какой-то (неизвестной) причине это
        // не удалось. Например, вследствие ошибки реализации Java 1.1.1
        // под Linux удаление каталога не удается никогда.
        if (!success) fail("Delete: удаление не удалось");
    }

    /** Вспомогательный метод, генерирующий исключение */
    protected static void fail(String msg) throws IllegalArgumentException {
        throw new IllegalArgumentException(msg);
    }
}
```

Копирование содержимого файла

Пример 3.2 показывает программу, копирующую содержимое заданного файла в другой файл. Эта программа использует класс `File` почти так же, как это делалось в примере 3.1: для проверки наличия исходного файла, а также того, что файл, в который мы намерены копировать, допускает запись и т. д. Но здесь можно увидеть также применение потоков для работы с содержимым файлов. В этом примере используются класс `FileInputStream` для чтения байтов из исходного файла и класс `FileOutputStream` для копирования этих байтов в целевой файл.

Метод `copy()` реализует функциональное содержание программы. Этот метод хорошо прокомментирован, чтобы помочь вам проследить все совершаемые действия. Во-первых, он выполняет очень много проверок, чтобы убедиться в правомочности запроса на копирование. Если все проверки пройдены, он создает класс `FileInputStream` для чтения байтов из исходного файла и класс `FileOutputStream` для записи этих байтов в целевой файл. Обратите внимание на применение буфера в виде массива байтов, служащего для хранения байтов во время копирования. Особое внимание уделите короткому циклу `while`, который, собственно, и осуществляет копирование. Сочетание присваивания и проверки в разделе условия цикла `while` – это полезная конструкция, часто встречающаяся при программировании ввода/вывода. Обратите также внимание на оператор `finally`, который гарантирует, что все потоки должным образом закрыты до выхода из программы.

Эта программа применяет потоки не только для чтения из файлов и записи в них. Прежде чем начать переписывать существующий файл, этот пример запрашивает у пользователя подтверждение. Здесь показано, как следует читать строки текста при помощи класса `BufferedReader`, считывающего отдельные символы из класса `InputStreamReader`, считывающего, в свою очередь, байты из `System.in` (`InputStream`), который, наконец, сам считывает нажатия клавиш на пользовательской клавиатуре. Вдобавок программа отображает текстовый вывод при помощи потоков `System.out` и `System.err`, каждый из которых является экземпляром класса `PrintStream`.

Статический метод `FileCopy.copy()` может вызываться непосредственно любой программой. Класс `FileCopy` содержит также метод `main()`, так что он может использоваться и как самостоятельная программа.

Пример 3.2. FileCopy.java

```
package com.davidflanagan.examples.io;
import java.io.*;

/**
 * Этот класс является самостоятельной программой для копирования файлов
 * и определяет также статический метод copy(), который могут использовать
 * для копирования файлов другие программы.
 **/
```

```
public class FileCopy {
    /** Метод main() самостоятельной программы. Вызывает метод copy(). */
    public static void main(String[] args) {
        if (args.length != 2)    // Проверка аргументов
            System.err.println("Формат: java FileCopy <исходный файл>
                                    <конечный файл>");
        else {
            // Вызывается метод copy() для осуществления копирования;
            // отображаются все сообщения об ошибках
            try { copy(args[0], args[1]); }
            catch (IOException e) { System.err.println(e.getMessage()); }
        }
    }

    /**
     * Статический метод, фактически производящий копирование файла.
     * До копирования файла, однако, он совершает множество проверок,
     * чтобы убедиться, что все обстоит надлежащим образом.
     */
    public static void copy(String from_name, String to_name)
        throws IOException
    {
        File from_file = new File(from_name);  // Из объектов String получаем
                                               // объекты File
        File to_file = new File(to_name);

        // Сначала убеждаемся, что исходный файл существует,
        // является файлом и доступен для чтения.
        if (!from_file.exists())
            abort("нет такого исходного файла: " + from_name);
        if (!from_file.isFile())
            abort("невозможно копирование каталога: " + from_name);
        if (!from_file.canRead())
            abort("исходный файл не доступен для чтения: " + from_name);

        // Если заданный «конечный файл» - это каталог, в качестве имени
        // конечного файла используется имя исходного файла
        if (to_file.isDirectory())
            to_file = new File(to_file, from_file.getName());

        // Если файл с заданным именем конечного файла существует, убеждаемся,
        // что он доступен для записи, и, прежде чем его перезаписывать,
        // запрашиваем подтверждение пользователя. Если конечный файл
        // не существует, убеждаемся, что такой каталог существует
        // и доступен для записи.
        if (to_file.exists()) {
            if (!to_file.canWrite())
                abort("конечный файл не доступен для записи: " + to_name);
            // Спрашиваем, следует ли перезаписать существующий файл
            System.out.print("Перезаписать существующий файл " + to_file.
                                getName() + "? (Y/N): ");
            System.out.flush();
            // Получаем ответ пользователя
```

```
            BufferedReader in=
                new BufferedReader(new InputStreamReader(System.in));
            String response = in.readLine();
            // Проверяем ответ пользователя. Если это не Yes,
            // копирование отменяется.
            if (!response.equals("Y") && !response.equals("y"))
                abort("существующий файл не был перезаписан.");
        }
        else {
            // Если файла с заданным именем нет, проверяем, существует ли
            // каталог с таким именем и доступен ли он для записи.
            // Если getParent() возвращает null, каталогом будет текущий dir.,
            // для определения того, что это за каталог, применяется
            // системное свойство user.dir.
            String parent = to_file.getParent();  // Каталог назначения
            if (parent == null)     // Если такового нет, используется
                                    // текущий каталог
                parent = System.getProperty("user.dir");
            File dir = new File(parent);          // Преобразуем его в файл.
            if (!dir.exists())
                abort("каталог назначения не существует: "+parent);
            if (dir.isFile())
                abort("каталог назначения не является каталогом: " + parent);
            if (!dir.canWrite())
               abort("каталог назначения не доступен для записи: " + parent);
        }

        // Если мы добрались до этого места, значит, все в порядке. Так что
        // начинаем копировать файл, за один прием перемещая буфер байтов.
        FileInputStream from = null;  // Поток для чтения из исходного файла
        FileOutputStream to = null;   // Поток для записи в конечный файл
        try {
            from = new FileInputStream(from_file);  // Создается поток ввода
            to = new FileOutputStream(to_file);     // Создается поток вывода
            byte[] buffer = new byte[4096];  // Для хранения содержимого файла
            int bytes_read;                  // Число байтов в буфере

            // В буфер считывается порция байтов, затем они выводятся,
            // цикл исполняется, пока мы не достигнем конца файла (пока read()
            // не возвратит -1).  Обратите внимание на сочетание присваивания
            // и сравнения в цикле while.  Это обычная конструкция при
            // программировании ввода/вывода.
            while((bytes_read = from.read(buffer)) != -1)  // Читаем до
                                                           // достижения EOF
                to.write(buffer, 0, bytes_read);           // Записываем
        }
        // Всегда закрываем потоки, даже если были выданы исключения
        finally {
            if (from != null) try { from.close(); } catch (IOException e) { ;
}
            if (to != null) try { to.close(); } catch (IOException e) { ; }
        }
    }
```

```
    /** Вспомогательный метод, генерирующий исключения */
    private static void abort(String msg) throws IOException {
        throw new IOException("FileCopy: " + msg);
    }
}
```

Чтение и отображение текстовых файлов

Пример 3.3 демонстрирует класс `FileViewer`. Он соединяет применение класса `File` и потоков ввода/вывода для чтения содержимого текстового файла с приемами GUI для отображения его содержимого. `FileViewer` применяет компонент `java.awt.TextArea` для отображения содержимого файла, как показано на рис. 3.1. Пример 3.3 использует методики графического интерфейса пользователя, представленные в главе 10 «Графические интерфейсы пользователя (GUI)». Если вы еще не прочли эту главу или не имеете опыта AWT-программирования, вы, вероятно, поймете не весь код примера. Это не беда; сосредоточьтесь на коде, осуществляющем ввод/вывод, которому посвящена настоящая глава.

Конструктор `FileViewer` занят, главным образом, вопросами настройки нужного GUI. В конце конструктора, однако, можно найти некоторые интересные применения объекта `File`. Сердцевиной этого примера является метод `setFile()`. Именно в нем содержимое файла загружается

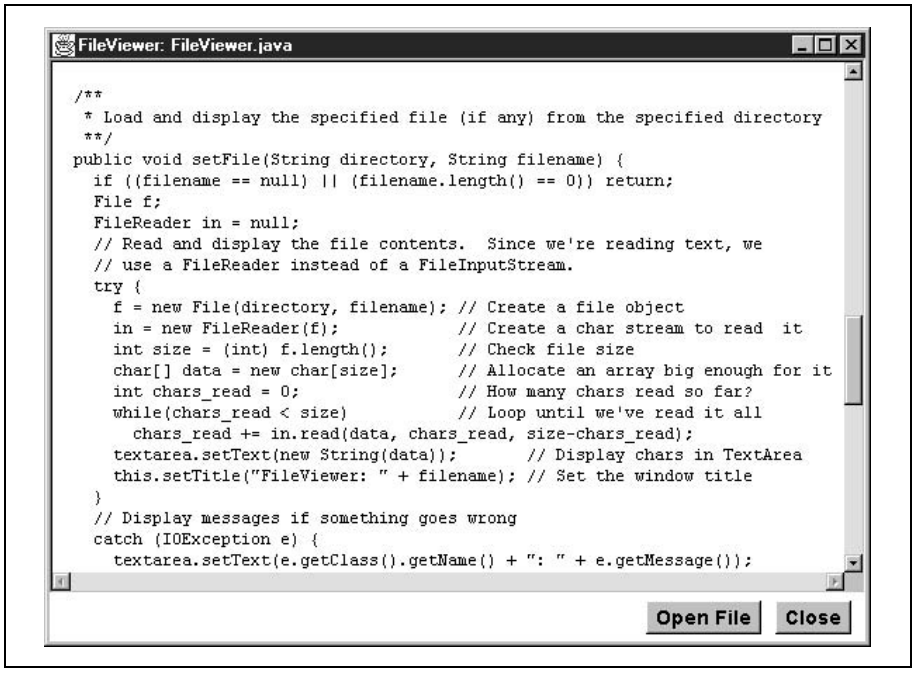


Рис. 3.1. *Окно FileViewer*

и отображается. Поскольку содержимое файла должно отображаться компонентом TextArea, законно будет предположить, что в файле содержатся символы. То есть вместо байтового потока ввода, использовавшегося в программе FileCopy примера 3.2, следует использовать символьный поток ввода FileReader 3. И вновь следует воспользоваться оператором finally, чтобы гарантировать правильное закрытие потока FileReader.

Метод actionPerformed() обрабатывает события GUI. Если пользователь щелкает мышью по кнопке **Open File**, этот метод создает объект FileDialog, позволяющий пользователю указать новый отображаемый файл. Обратите внимание на то, как устанавливается каталог по умолчанию до отображения диалогового окна и как он устанавливается снова после того, как пользователь сделал выбор. Это оказывается возможным, поскольку метод show() в действительности блокируется до тех пор, пока пользователь не выберет файл и не закончит диалог.

Класс FileViewer разработан для использования другими классами. У него есть, однако, и собственный метод main(), так что он может запускаться как самостоятельная программа.

Пример 3.3. FileViewer.java

```
package com.davidflanagan.examples.io;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

/**
 * Этот класс создает и отображает окно, содержащее область TextArea,
 * в которой визуализируется содержимое текстового файла.
 **/
public class FileViewer extends Frame implements ActionListener {
    String directory;  // Каталог, принимаемый по умолчанию FileDialog
    TextArea textarea; // Область, в которой будет отображаться содержимое файла

    /** Вспомогательный конструктор: окно просмотра открывается пустым */
    public FileViewer() { this(null, null); }
    /** Вспомогательный конструктор: отображается файл из текущего каталога */
    public FileViewer(String filename) { this(null, filename); }

    /**
     * Настоящий конструктор. Создает объект FileViewer для отображения
     * заданного файла из заданного каталога
     **/
    public FileViewer(String directory, String filename) {
        super();  // Создается рамка

        // По требованию пользователя окно уничтожается
        addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) { dispose(); }
            });
        // Создается область TextArea для отображения в ней содержимого файла
```

```
        textarea = new TextArea("", 24, 80);
        textarea.setFont(new Font("MonoSpaced", Font.PLAIN, 12));
        textarea.setEditable(false);
        this.add("Center", textarea);

        // Создается нижняя панель для двух кнопок
        Panel p = new Panel();
        p.setLayout(new FlowLayout(FlowLayout.RIGHT, 10, 5));
        this.add(p, "South");

        // Создаются две кнопки, и организуется обработка их нажатий
        Font font = new Font("SansSerif", Font.BOLD, 14);
        Button openfile = new Button("Open File");
        Button close = new Button("Close");
        openfile.addActionListener(this);
        openfile.setActionCommand("open");
        openfile.setFont(font);
        close.addActionListener(this);
        close.setActionCommand("close");
        close.setFont(font);
        p.add(openfile);
        p.add(close);

        this.pack();

        // Определяется каталог на основании filename или, если необходимо,
        // текущего каталога (dir)
        if (directory == null) {
            File f;
           if ((filename != null) && (f = new File(filename)).isAbsolute()) {
                directory = f.getParent();
                filename = f.getName();
            }
            else directory = System.getProperty("user.dir");
        }

        this.directory = directory;  // Каталог запоминается
                                     // для нужд FileDialog
       setFile(directory, filename); // Теперь файл загружается и отображается
    }

    /**
     * Загружается и отображается заданный файл из заданного каталога
     **/
    public void setFile(String directory, String filename) {
        if ((filename == null) || (filename.length() == 0)) return;
        File f;
        FileReader in = null;
        // Считывается и отображается содержимое файла. Поскольку читается
        // текст, применяется поток FileReader вместо FileInputStream.
        try {
            f = new File(directory, filename); // Создаются объект File
           in = new FileReader(f);        // и символьный поток для его чтения
            char[] buffer = new char[4096];    // За один прием считывается
```

```
                                                // 4K символов
        int len;                    // Число символов, считанных на этот раз
         textarea.setText("");       // Очистка textarea
        while((len = in.read(buffer)) != -1) { // Считывание порции символов
             String s = new String(buffer, 0, len); // Преобразование
                                                    // в строку
             textarea.append(s);                    // И отображение строк
         }
         this.setTitle("FileViewer: " + filename);  // Устанавливается
                                                    // заголовок окна
         textarea.setCaretPosition(0);         // Переход к началу файла
     }
     // Отображаем сообщения, если что-то не в порядке
     catch (IOException e) {
         textarea.setText(e.getClass().getName() + ": " + e.getMessage());
         this.setTitle("FileViewer: " + filename + ":
                        Исключение ввода/вывода ");
     }
     // Всегда следует обеспечивать закрытие потока ввода!
    finally { try { if (in!=null) in.close(); } catch (IOException e) {} }
}

/**
 * Обработка нажатий кнопок
 **/
public void actionPerformed(ActionEvent e) {
    String cmd = e.getActionCommand();
    if (cmd.equals("open")) {      // Если пользователь щелкнул
                                   // по кнопке "Open"
        // Создается диалоговое окно, предлагающее задать новый файл,
        // который следует отобразить
       FileDialog f = new FileDialog(this, "Open File", FileDialog.LOAD);
       f.setDirectory(directory);   // Устанавливается каталог по умолчанию

        // Отображается диалоговое окно и ожидается ответ пользователя
        f.show();

        directory = f.getDirectory();    // Запоминается принимаемый
                                         // по умолчанию каталог
        setFile(directory, f.getFile()); // Загружается и отображается
                                         // выбранный файл
        f.dispose();                     // Закрывается диалоговое окно
    }
    else if (cmd.equals("close"))     // Если пользователь щелкнул
                                      // по кнопке "Close",
        this.dispose();               //окно закрывается
}

/**
 * Метод FileViewer может использоваться другими классами или
 * применяться самостоятельно за счет следующего метода main().
 **/
static public void main(String[] args) throws IOException {
```

```
            // Создается объект FileViewer
            Frame f = new FileViewer((args.length == 1)?args[0]:null);
            // Подготовка к выходу по закрытию окна FileViewer
            f.addWindowListener(new WindowAdapter() {
                    public void windowClosed(WindowEvent e) { System.exit(0); }
                });
            // Открытие окна
            f.show();
        }
    }
```

Содержимое каталога и информация о файле

Подобно тому как класс `FileViewer` из примера 3.3 отображает содержимое файла при помощи компонента `TextArea`, класс `FileLister`, показанный в примере 3.4, отображает содержимое каталога при помощи компонента `java.awt.List`. Когда вы выбираете в списке имя файла или каталога, программа отображает информацию о файле или каталоге (размер, дату последнего изменения и т. д.) при помощи компонента `TextField`. Когда вы делаете двойной щелчок на каталоге, отображается содержимое этого каталога. Когда вы делаете двойной щелчок на файле, отображается содержимое этого файла при помощи объекта `FileViewer`. На рис. 3.2 показано окно `FileLister`. Опять же, если вы еще не знакомы с GUI-программированием на Java, не пытайтесь понять все в приведенном коде, пока не прочтете главу 10; вместо этого уделите внимание различным способам применения объекта `File`, показанным в этом примере.

Значительную часть этого примера составляет GUI-механика, заставляющая работать `FileLister`. Метод `listDirectory()` выводит список содержимого каталога, используя необязательно задаваемый объект `FilenameFilter`, который передается конструктору `FileLister()`. Этот объект определяет метод `accept()`, который вызывается для каждого элемента списка, с тем чтобы определить, следует ли его отображать.

Метод `itemStateChanged()` вызывается, когда выбран элемент списка. Он получает информацию о файле или каталоге и отображает ее. Метод `actionPerformed()` — это еще один метод, ожидающий наступления событий. Он вызывается, когда пользователь либо щелкнул мышью по кнопкам (объектам `Button`), либо сделал двойной щелчок на элементе списка. Если пользователь сделал двойной щелчок на каталоге, программа выводит список его содержимого. Если же пользователь сделал двойной щелчок на файле, она создает и отображает окно `FileViewer` для визуализации содержимого файла.

Подобно классу `FileViewer`, класс `FileLister` может использоваться другими классами или вызываться как самостоятельная программа. Если вы вызываете его в виде автономной программы, он отображает содержимое текущего каталога. Также его можно вызвать с заданным име-

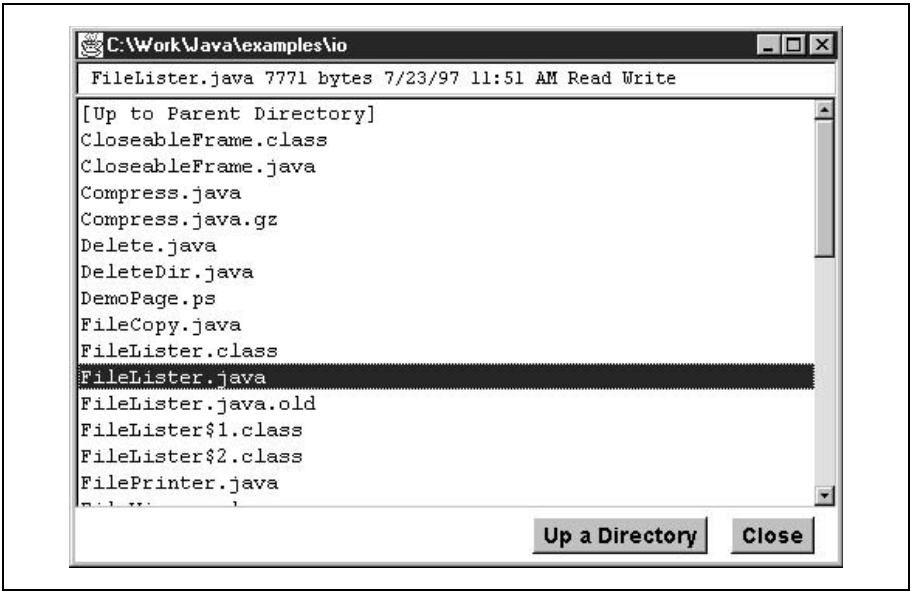


Рис. 3.2. *Окно FileLister*

нем каталога для показа содержимого этого каталога. Если задать необязательный флаг `-e`, указав после него расширение имени файла, программа отфильтрует список файлов и отобразит только файлы с указанным расширением. Обратите внимание на то, как метод `main()` анализирует аргументы, передаваемые в командной строке, и применяет безымянный (anonimous) класс для реализации интерфейса `FilenameFilter`.

Пример 3.4. FileLister.java

```
package com.davidflanagan.examples.io;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.text.DateFormat;
import java.util.Date;

/**
 * Этот класс создает и отображает окно, содержащее список
 * файлов и подкаталогов заданного каталога. Щелчок на элементе списка
 * вызывает отображение дополнительной информации о нем.
 * Двойной щелчок вызывает отображение содержимого файла или каталога.
 * Если задан фильтр имен, FilenameFilter фильтрует отображаемый список.
 **/
public class FileLister extends Frame реализует ActionListener, ItemListener {
    private List list;          // Для отображения в нем содержимого каталога
    private TextField details;   // Для отображения в нем подробной информации
```

```
    private Panel buttons;              // В панели находятся кнопки
    private Button up, close;           // Кнопки Up и Close
    private File currentDir;            // Текущий отображаемый каталог
    private FilenameFilter filter;      // Необязательный фильтр для каталога
    private String[] files;             // Содержимое каталога
    private DateFormat dateFormatter =  // Для правильного отображения даты
                                        // и времени дня
          DateFormat.getDateTimeInstance(DateFormat.SHORT, DateFormat.SHORT);

    /**
     * Конструктор: создает GUI и отображает начальный каталог.
     **/
    public FileLister(String directory, FilenameFilter filter) {
        super("File Lister");           // Создается окно
        this.filter = filter;           // Сохраняется фильтр, если он задан

        // По требованию пользователя уничтожается окно
        addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) { dispose(); }
            });

        list = new List(12, false);         // Создается список
        list.setFont(new Font("MonoSpaced", Font.PLAIN, 14));
        list.addActionListener(this);
        list.addItemListener(this);

        details = new TextField();          // Создается область отображения
                                            // подробной информации о файле
        details.setFont(new Font("MonoSpaced", Font.PLAIN, 12));
        details.setEditable(false);

        buttons = new Panel();              // Установка панели кнопок
        buttons.setLayout(new FlowLayout(FlowLayout.RIGHT, 15, 5));
        buttons.setFont(new Font("SansSerif", Font.BOLD, 14));

        up = new Button("Up a Directory"); // Установка двух кнопок
        close = new Button("Close");
        up.addActionListener(this);
        close.addActionListener(this);

        buttons.add(up);                // Добавление кнопок к панели
        buttons.add(close);

        this.add(list, "Center");       // Заполнение окна содержимым
        this.add(details, "North");
        this.add(buttons, "South");
        this.setSize(500, 350);

        listDirectory(directory);       // А теперь отображается содержимое
                                        // начального каталога.
    }

    /**
     * Этот метод использует метод list() для получения всех элементов
     * каталога и затем отображает их при помощи компонента List.
```

```
     **/
    public void listDirectory(String directory) {
        // Строка преобразуется в объект File и выполняется проверка,
        // существует ли такой каталог
        File dir = new File(directory);
        if (!dir.isDirectory())
           throw new IllegalArgumentException("FileLister:
                                              нет такого каталога");

        // Принимается (отфильтрованное) содержимое каталога
        files = dir.list(filter);

        // Сортируется список имен файлов. До появления Java 1.2 можно было
        // воспользоваться для сортировки
        // методом com.davidflanagan.examples.classes.Sorter.sort().
        java.util.Arrays.sort(files);

        // Из списка удаляются все старые элементы и вставляются новые
        list.removeAll();
        list.add("[Up to Parent Directory]");  // Специальный элемент списка
        for(int i = 0; i < files.length; i++) list.add(files[i]);

        // В заголовке окна и в области подробной информации
        // отображается имя каталога
        this.setTitle(directory);
        details.setText(directory);

        // На будущее запоминается текущий каталог.
        currentDir = dir;
    }

    /**
     * Этот метод интерфейса ItemListener использует различные методы
     * класса File для получения информации о файле или каталоге.
     * Затем он отображает эту информацию.
     **/
    public void itemStateChanged(ItemEvent e) {
        int i = list.getSelectedIndex() - 1;  // Минус 1 для элемента
                                              // Up To Parent (к родительскому
                                              // каталогу)
        if (i < 0) return;
        String filename = files[i];         // Принимается выбранный элемент
        File f = new File(currentDir, filename);  // Преобразуется в File
        if (!f.exists())                  // Подтверждается его существование
            throw new IllegalArgumentException("FileLister: " +
                                              "нет такого файла или каталога");

        // Получение подробной информации о файле или каталоге,
        // соединение ее в одну строку
        String info = filename;
        if (f.isDirectory()) info += File.separator;
        info += " " + f.length() + " байт ";
        info += dateFormatter.format(new java.util.Date(f.lastModified()));
        if (f.canRead()) info += " для чтения";
```

```
    if (f.canWrite()) info += " для записи";

    // И отображение строки с подробностями
    details.setText(info);
}

/**
 * Этот метод интерфейса ActionListener вызывается, когда пользователь
 * делает двойной щелчок на элементе или нажимает одну из кнопок.
 * Если двойной щелчок сделан на файле, создается FileViewer
 * для отображения этого файла. Если двойной щелчок сделан на каталоге,
 * вызывается метод ListDirectory() для отображения этого каталога
 **/
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == close) this.dispose();
    else if (e.getSource() == up) { up(); }
    else if (e.getSource() == list) {  // Двойной щелчок на элементе
        int i = list.getSelectedIndex(); // На каком элементе?
       if (i == 0) up();   // Обрабатывается первый элемент, Up To Parent
        else {              // В противном случае принимается имя файла
            String name = files[i-1];
           File f = new File(currentDir, name);    // Преобразуется в File
            String fullname = f.getAbsolutePath();
           if (f.isDirectory()) listDirectory(fullname); // Отображается
                                                  // содержимое каталога
            else new FileViewer(fullname).show();       // Отображается
                                                    // содержимое файла
        }
    }
}

/** Вспомогательный метод для отображения содержимого
    родительского каталога */
protected void up() {
    String parent = currentDir.getParent();
    if (parent == null) return;
    listDirectory(parent);
}

/** Вспомогательный метод, используемый методом main() */
public static void usage() {
    System.out.println("Формат: java FileLister [имя каталога] " +
                       "[-e расширение файла]");
    System.exit(0);
}

/**
 * Метод main(), обеспечивающий самостоятельный запуск FileLister. Анали-
 * зируются аргументы, передаваемые в командной строке, и создается объект
 *  FileLister. Если задано расширение, для него создается FilenameFilter.
 *  Если никакой каталог не задан, используется текущий каталог.
 **/
public static void main(String args[]) throws IOException {
```

```
        FileLister f;
        FilenameFilter filter = null;  // Фильтр на случай, если он задан
        String directory = null;       // Заданный или текущий каталог

        // Цикл по массиву аргументов и их анализ
        for(int i = 0; i < args.length; i++) {
            if (args[i].equals("-e")) {
                if (++i >= args.length) usage();
                final String suffix = args[i]; // Переменная объявлена
                                               // как final для возможности ее
                                                 // использования в описанном
                                                 // ниже безымянном классе

                // Этот класс является простым фильтром имен файлов
                // FilenameFilter.  Он определяет метод accept(), необходимый
                // для выяснения того, следует ли отображать заданный файл.
                // Файл будет отображен, если его имя оканчивается
                // заданным расширением или если он является каталогом.
                filter = new FilenameFilter() {
                        public boolean accept(File dir, String name) {
                            if (name.endsWith(suffix)) return true;
                            else return (new File(dir, name)).isDirectory();
                        }
                    };
            }
            else {
                if (directory != null) usage();  // Если уже задан, ошибка.
                else directory = args[i];
            }
        }

        // Если каталог не задан, используется текущий
        if (directory == null) directory = System.getProperty("user.dir");
        // Создается объект FileLister с заданным каталогом и фильтром.
        f = new FileLister(directory, filter);
        // Подготавливается завершение приложения при закрытии окна
        f.addWindowListener(new WindowAdapter() {
            public void windowClosed(WindowEvent e) { System.exit(0); }
        });
        // Наконец, выводится окно.
        f.show();
    }
}
```

Сжатие файлов и каталогов

Пример 3.5 демонстрирует интересное применение классов потоков: сжатие (compressing) файлов и каталогов. Интересующие нас в этом примере классы на самом деле не принадлежат пакету `java.io`**, а являются частью пакета** `java.util.zip`**. Класс** `Compress` **определяет два статических метода: метод** `gzipFile()`**, который сжимает файл, применяя**

формат сжатия GZIP, и метод `zipDirectory()`, который сжимает файлы (но не каталоги) в каталог, применяя формат архивирования и сжатия ZIP. Метод `gzipFile()` использует класс `GZIPOutputStream`, тогда как метод `zipDirectory()` использует классы `ZipOutputStream` и `ZipEntry`, все из пакета `java.util.zip`.

Этот пример демонстрирует многообразные возможности классов потоков и еще раз показывает, как потоки могут вкладываться друг в друга, так что вывод одного потока становится вводом для другого. Этот прием позволяет достигать разнообразных эффектов. И снова обратите внимание на цикл `while`, в обоих методах осуществляющий фактическое копирование данных из исходного в сжатый файл. Эти методы не пытаются обрабатывать исключения; вместо этого они просто передают их вызывающей программе, как это обычно и следует делать.

Класс `Compress` предназначен для использования другими программами, поэтому он сам не содержит метода `main()`. Однако пример содержит внутренний класс `Compress.Test`, в котором есть метод `main()`. Его можно использовать для тестирования методов `gzipFile()` и `zipDirectory()`.

Пример 3.5. Compress.java

```
package com.davidflanagan.examples.io;
import java.io.*;
import java.util.zip.*;

/**
 * Этот класс определяет два статических метода для преобразования файлов
 * по алгоритму GZIP и для преобразования содержимого каталога
 * по алгоритму ZIP. Он также определяет демонстрационную программу
 * в виде вложенного класса.
 **/
public class Compress {
    /** Сжимаем по алгоритму GZIP содержимое файла from и сохраняем
        его в файле to. */
    public static void gzipFile(String from, String to) throws IOException {
        // Создаем поток для чтения из файла from.
        FileInputStream in = new FileInputStream(from);
        // Создаем поток для сжатия данных и записи их в файл to.
        GZIPOutputStream out = new GZIPOutputStream(new
FileOutputStream(to));
        // Копируем байты из одного потока в другой
        byte[] buffer = new byte[4096];
        int bytes_read;
        while((bytes_read = in.read(buffer)) != -1)
            out.write(buffer, 0, bytes_read);
        // И закрываем потоки
        in.close();
        out.close();
    }
```

```
/** Сжимаем по алгоритму ZIP содержимое каталога и сохраняем
    его в файле zipfile */
public static void zipDirectory(String dir, String zipfile)
    throws IOException, IllegalArgumentException {
    // Проверяем, является ли каталог каталогом,
    // и принимаем его содержимое
    File d = new File(dir);
    if (!d.isDirectory())
        throw new IllegalArgumentException("Compress: не каталог:
                                           " + dir);
    String[] entries = d.list();
    byte[] buffer = new byte[4096];  // Создаем буфер для копирования
    int bytes_read;

    // Создаем поток для сжатия данных и записываем их в zipfile
    ZipOutputStream out =
        new ZipOutputStream(new FileOutputStream(zipfile));

    // Цикл по всем элементам каталога
    for(int i = 0; i < entries.length; i++) {
        File f = new File(d, entries[i]);
        if (f.isDirectory()) continue;        // Пропускаем подкаталоги
        FileInputStream in = new FileInputStream(f); // Поток для чтения
                                                     // файла
        ZipEntry entry = new ZipEntry(f.getPath());  // Создаем элемент
                                                     // архива ZipEntry
       out.putNextEntry(entry);              // Сохраняем элемент архива
        while((bytes_read = in.read(buffer)) != -1)  // Копируем байты
            out.write(buffer, 0, bytes_read);
       in.close();                                   // Закрываем поток ввода
    }
    // Когда цикл завершен, закрываем поток вывода
    out.close();
}

/**
 * Этот вложенный класс является тестовой программой, демонстрирующей
 * применение статических методов, определенных выше.
 **/
public static class Test {
    /**
     * Сжимаем заданный файл или каталог. Если для архива имя не задано,
     * к имени сжимаемого файла присоединяется .gz, а к имени
     * архивируемого каталога - .zip
     **/
    public static void main(String args[]) throws IOException {
        if ((args.length != 1) && (args.length != 2)) {  // Проверяем
                                                         // аргументы
           System.err.println("Формат: java Compress$Test <from> [<to>]");
            System.exit(0);
        }
        String from = args[0], to;
```

```
            File f = new File(from);
            boolean directory = f.isDirectory();  // Файл это или каталог?
            if (args.length == 2) to = args[1];
            else {                                // Если имя архива не задано,
                if (directory) to = from + ".zip"; // используем суффикс .zip
                else to = from + ".gz";           // или суффикс .gz
            }

            if ((new File(to)).exists()) {   // Убеждаемся, что не пере-
                                             // записываем существующий файл
                System.err.println("Compress: не перезаписываем
                                   существующие файлы: "+ to);
                System.exit(0);
            }

            // Наконец, для выполнения задачи вызываем один
            // из определенных выше методов.
            if (directory) Compress.zipDirectory(from, to);
            else Compress.gzipFile(from, to);
        }
    }
}
```

Фильтрация потоков символов

`FilterReader` — это абстрактный класс, определяющий нулевой (null) фильтр; он считывает символы из заданного объекта `Reader` и возвращает их без изменений. Другими словами, `FilterReader` определяет пустые реализации всех методов класса `Reader`. Его подкласс, чтобы осуществить какую бы то ни было фильтрацию, должен заместить по меньшей мере два метода `read()`. Некоторые подклассы могут замещать и другие методы. Пример 3.6 показывает класс `RemoveHTMLReader`, являющийся подклассом `FilterReader`, который читает HTML-текст из потока и отфильтровывает из возвращаемого текста все теги HTML.

В этом примере реализуется фильтрация тегов HTML в версии метода `read()`, принимающей три аргумента, а затем на основе этого более сложного метода реализуется версия без аргументов. Пример содержит внутренний класс `Test` с методом `main()`, показывающим, как можно использовать класс `RemoveHTMLReader`.

Обратите внимание на то, что вы можете определить также класс `RemoveHTMLWriter`, проделывая такую же фильтрацию в подклассе `FilterWriter`. Или чтобы фильтровать байты, а не символы, можно создать подклассы потоков `FilterInputStream` и `FilterOutputStream`. Класс `RemoveHTMLReader` — это только один пример потока с фильтрованием. Можно создать потоки, которые подсчитывают число обработанных символов или байтов, преобразуют все символы к верхнему регистру, выделяют URL, выполняют операции поиска и замены, преобразуют символы конца строки из принятых в Unix `LF` в принятые в Windows `CRLF` и т. д.

Пример 3.6. RemoveHTMLReader.java

```
package com.davidflanagan.examples.io;
import java.io.*;

/**
 * Простой FilterReader, вырезающий из потока символов HTML-теги
 * (или все, что содержится между двумя угловыми скобками).
 **/
public class RemoveHTMLReader extends FilterReader {
    /** Тривиальный конструктор. Просто инициализирует наш базовый класс */
    public RemoveHTMLReader(Reader in) { super(in); }

    boolean intag = false;    // Используется для запоминания того,
                              // что мы находимся «внутри» тега

    /**
     * Это реализация пустого метода read() класса FilterReader.
     * Он вызывает in.read()для заполнения буфера символами, а затем вырезает
     * теги HTML. (in – это защищенное (protected) поле базового класса).
     **/
    public int read(char[] buf, int from, int len) throws IOException {
        int numchars = 0;         // Число считанных символов
        // Цикл здесь нужен, потому что, считав порцию символов,
        // мы могли бы вырезать их все, и нечего было бы возвращать.
        while (numchars == 0) {
            numchars = in.read(buf, from, len); // Считываем символы
            if (numchars == -1) return -1;      // Проверка на конец файла
                                                // и ее обработка.

           // Цикл по всем считанным символам, вырезающий теги HTML. Символы,
            // не входящие в теги, копируются поверх предшествующих тегов
            int last = from;            // Индекс последнего не-HTML-символа
            for(int i = from; i < from + numchars; i++) {
                if (!intag) {           // Если «вне» тега HTML
                  if (buf[i] == '<') intag = true; // проверка на начало тега
                   else buf[last++] = buf[i];       // и копирование символа
                }
              else if (buf[i] == '>') intag = false;  // проверка на конец тега
            }
            numchars = last - from; // Вычисляется количество оставшихся
                                    // символов
        }                           // И если оно больше нуля,
        return numchars;            // это число возвращается.
    }

    /**
     * Это еще один пустой метод read(), который необходимо реализовать.
     * Он реализуется с помощью вышеприведенного метода. Наш базовый класс
     * реализует остальные методы read() в терминах этих двух.
     **/
    public int read() throws IOException {
        char[] buf = new char[1];
        int result = read(buf, 0, 1);
```

```
            if (result == -1) return -1;
            else return (int)buf[0];
        }

        /** Этот класс определяет метод main(), тестирующий
            класс RemoveHTMLReader */
        public static class Test {
            /** Тестовая программа: считываем текстовый файл, вырезаем
                разметку HTML, печатаем на консоль */
            public static void main(String[] args) {
                try {
                    if (args.length != 1)
                        throw new IllegalArgumentException("Неправильное
                                                           число аргументов");
                    // Создаем поток для чтения из файла и вырезания тегов
                    BufferedReader in = new BufferedReader(
                                new RemoveHTMLReader(new FileReader(args[0])));
                    // Читаем построчно, распечатываем строки на консоли
                    String line;
                    while((line = in.readLine()) != null)
                        System.out.println(line);
                    in.close();  // Закрываем поток.
                }
                catch(Exception e) {
                    System.err.println(e);
                    System.err.println("Формат: java RemoveHTMLReader$Test" +
                                       " <имя_файла>");
                }
            }
        }
    }
```

Фильтрация строк текста

Пример 3.7 определяет класс `GrepReader`, еще один специализированный поток ввода. Этот поток считывает строки текста из заданного потока `Reader` и возвращает только те строки, которые содержат заданную подстроку. Таким образом, он работает как команда *fgrep* системы Unix, выполняя «grep», то есть поиск. Класс `GrepReader` производит фильтрацию, но он фильтрует текст построчно, а не посимвольно, поэтому он является подклассом класса `BufferedReader`, а не `FilterReader`.

Коды этого примера совершенно прозрачны. Пример содержит внутренний класс `Test` с методом `main()`, демонстрирующим применение потока `GrepReader`. Эту тестовую программу можно вызвать так:

```
% java com.davidflanagan.examples.io.GrepReader\$Test needle haystack.txt
```

Пример 3.7. GrepReader.java

```
package com.davidflanagan.examples.io;
```

```
import java.io.*;

/**
 * Этот подкласс класса BufferedReader отфильтровывает все строки текста,
 * не содержащие заданный образец.
 **/
public class GrepReader extends BufferedReader {
    String pattern;  // Строка, совпадения с которой мы будем отыскивать.

    /** Передаем строку базовому классу и сами запоминаем ее */
    public GrepReader(Reader in, String pattern) {
        super(in);
        this.pattern = pattern;
    }

    /**
     * Это наш фильтр: вызывается метод базового класса readLine()
     * для получения строк, но возвращаются только строки, содержащие
     * заданный образец. Когда метод базового класса readLine() возвращает
     * null (конец файла), мы также возвращаем null.
     **/
    public final String readLine() throws IOException {
        String line;
        do { line = super.readLine(); }
        while ((line != null) && line.indexOf(pattern) == -1);
        return line;
    }

    /**
     * Этот класс демонстрирует применение класса GrepReader.
     * Он печатает строки файла, содержащие заданную подстроку.
     **/
    public static class Test {
        public static void main(String args[]) {
            try {
                if (args.length != 2)
                    throw new IllegalArgumentException("Неправильное
                                                        число аргументов");
               GrepReader in=new GrepReader(new FileReader(args[1]), args[0]);
                String line;
               while((line = in.readLine()) != null) System.out.println(line);
                in.close();
            }
            catch (Exception e) {
                System.err.println(e);
                System.out.println("Формат: java GrepReader$Test" +
                                   " <образец> <файл>");
            }
        }
    }
}
```

Специализированный поток вывода HTML

Широкое применение языка Java началось, в том числе, с запуска в броузерах апплетов – миниатюрных программ, распространяемых по Интернету. (Мы будем обсуждать апплеты подробнее в главе 15 «Апплеты».) Поскольку апплеты запускаются в броузерах, которые являются мощным средством отображения HTML-документов, логично было бы предположить наличие какого-то способа использовать изнутри апплета способность броузера отображать HTML-документы.

Пример 3.8 показывает специализированный поток вывода `HTMLWriter`, обеспечивающий именно эту возможность. Когда новый поток `HTMLWriter` создан, он обращается к броузеру при помощи команд JavaScript и приказывает броузеру открыть новое окно. После записи текста в формате HTML в этот поток текст при помощи JavaScript передается в новое окно, которое анализирует и отображает полученный текст. Этот класс основан на технологии LiveConnect, разработанной фирмой Netscape, и классе `netscape.javascript.JSObject`. Класс `HTMLWriter` создавался для использования с Netscape Navigator 4.0 или более поздних версий, хотя компания Microsoft реализовала вариант этой технологии, и этот пример должен работать также с последними версиями Internet Explorer.

Обратите внимание на реализацию методов `write()` и `close()` и пустую реализацию метода `flush()`. Это три абстрактных метода класса `Writer`, которые должны быть реализованы в каждом конкретном подклассе. Обратите внимание также на метод `closeWindow()`, который добавлен в этот класс.

Я не жду от вас понимания класса JSObject или команд JavaScript, посылаемых через него броузеру. Если вам интересно узнать о них побольше, рекомендую мою книгу по JavaScript «JavaScript: The Definitive Guide», также опубликованную издательством O'Reilly.[1]

Пример содержит внутренний класс апплета `Test`, демонстрирующий применение класса `HTMLWriter`. Этот апплет считывает URL, указанный в параметрах апплета (то есть в теге `<PARAM>`), и создает поток `HTMLWriter` для отображения содержимого этого URL. Этот апплет можно протестировать при помощи, допустим, такого тега HTML:

```
<APPLET CODE="HTMLWriter$Test.class" WIDTH=10 HEIGHT=10 MAYSCRIPT>
  <PARAM NAME="url" VALUE="HTMLWriter.java">
</APPLET>
```

Обратите внимание на атрибут `MAYSCRIPT` тега `<APPLET>`. Он разрешает апплету Java вызывать команды JavaScript броузера. Тег `<PARAM>` зада-

[1] Дэвид Флэнаган «JavaScript. Подробное руководство», 4-е издание, Символ-Плюс, IV кв. 2003 г.

ет значение параметра `url` – URL-адрес файла, который апплет должен отобразить.

Для компиляции этого примера необходимо наличие у вас в пути `CLASSPATH` класса `netscape.javascript.JSObject`. Обычно этот класс можно найти в файле вроде *java/classes/java40.jar* в каталоге, где установлен Netscape. Поскольку этот пример предназначен для запуска в виде апплета в броузере, он должен иметь возможность (после его запуска) автоматически найти класс `JSObject`.

Еще один похожий специализированный поток вывода можно найти в примере 12.3 в главе 12 «Печать».

Пример 3.8. HTMLWriter.java

```
package com.davidflanagan.examples.io;
import java.io.*;
import java.net.*;
import java.applet.Applet;
import netscape.javascript.JSObject;    // Нужный нам специальный класс

/**
 * Поток вывода, посылающий HTML-текст во вновь созданное окно броузера.
 * Для передачи команд JavaScript броузеру он полагается на класс netscape.
 * javascript.JSObject и будет работать только в апплетах, запускаемых в
 * броузерах Netscape Navigator.
 **/
public class HTMLWriter extends Writer {
    JSObject main_window;        // начальное окно броузера
    JSObject window;             // создаваемое нами новое окно
    JSObject document;           // документ для этого нового окна
    static int window_num = 0;   // используется для присвоения
                                 // каждому новому окну уникального имени

    /**
     * При создании нового объекта HTMLWriter он открывает новое пустое окно
     * броузера для отображения своего вывода. Необходимо задать апплет
     * (это определяет главное окно броузера) и желательные размеры нового окна.
     **/
    public HTMLWriter(Applet applet, int width, int height) {
        // Убеждаемся, что можем найти необходимый нам класс JSObject.
        // Если это не так, выдаем предупреждение.
        try { Class c = Class.forName("netscape.javascript.JSObject"); }
        catch (ClassNotFoundException e) {
            throw new NoClassDefFoundError("HTMLWriter будет работать " +
                   "в Netscape Navigator 4.0 или более поздних версий " +
                  "или в броузерах, поддерживающих технологию LiveConnect");
        }

        // Принимаем у апплета ссылку на главное окно броузера.
        main_window = JSObject.getWindow(applet);

        // Создаем новое окно для отображения в нем вывода. Эта команда
        // посылает броузеру строку JavaScript
```

```
        window = (JSObject)
            main_window.eval("self.open('',"" +
                             "'HTMLWriter" + window_num++ + "',"" +
                             "'menubar,status,resizable,scrollbars,"" +
                             "width=" + width + ",height=" + height + "')");

        // Получаем объект Document для этого нового окна и открываем его.
        document = (JSObject) window.getMember("document");
        document.call("open", null);
    }

    /**
     * Это метод write(), необходимый во всех подклассах класса Writer.
     * Writer определяет через него все другие свои методы write().
     **/
    public void write(char[] buf, int offset, int length) {
        // Если нет окна или документа, не делается ничего.
        // Так будет, если поток закрыт или если код запущен не в Navigator.
        if ((window == null) || (document == null)) return;
        // Если окно закрыто пользователем, не делается ничего
        if (((Boolean)window.getMember("closed")).booleanValue()) return;
        // В противном случае создаем строку из заданных байтов
        String s = new String(buf, offset, length);
        // и передаем ее методу document.write() JavaScript для вывода
        // HTML-документа
        document.call("write", new String[] { s });
    }

    /**
     * Не существует общего способа заставить JavaScript протолкнуть
     * задержавшийся вывод, поэтому этот метод не делает ничего.
     * Для проталкивания следует вывести тег <P> или какой-нибудь
     * другой тег HTML, вызывающий в выводе переход на новую строку.
     **/
    public void flush() {}

    /**
     * Когда поток закрывается, закрывается и объект Document JavaScript
     * (но окно пока не закрывается.)
     **/
    public void close() { document.call("close", null); document = null; }

    /**
     * Если окно броузера еще открыто, закрываем его.
     * Этот метод есть только в классе HTMLWriter.
     **/
    public void closeWindow() {
        if (document != null) close();
        if (!((Boolean)window.getMember("closed")).booleanValue())
            window.call("close", null);
        window = null;
    }

    /** Завершающий метод, закрывающий окно на тот случай,
        если мы забыли это сделать. */
```

```
    public void finalize() { closeWindow(); }
    /**
     * Этот вложенный класс является апплетом, демонстрирующим применение
     * HTMLWriter. Он читает данные, расположенные по URL-адресу, заданному в
     * его параметре url, и записывает их в поток HTMLWriter. Он будет
     * работать только в броузерах Netscape 4.0 и более поздних версий.
     * Ему требуется приблизительно такой тег <APPLET>:
     *   <APPLET CODE="HTMLWriter$Test.class" WIDTH=10 HEIGHT=10 MAYSCRIPT>
     *   <PARAM NAME="url" VALUE="HTMLWriter.java">
     *   </APPLET>
     * Обратите внимание на атрибут MAYSCRIPT. Он необходим, чтобы позволить
     * апплету вызывать JavaScript.
     **/
    public static class Test extends Applet {
        HTMLWriter out;
        /** Когда апплет запущен, читаем и отображаем заданный URL-адрес */
        public void init() {
            try {
                // Принимаем URL-адрес, заданный в теге <PARAM>
                URL url =
                  new URL(this.getDocumentBase(), this.getParameter("url"));
                // Создаем поток для чтения данных с этого URL
                Reader in = new InputStreamReader(url.openStream());
                // Создаем поток HTMLWriter для вывода
                out = new HTMLWriter(this, 400, 200);
                // Считываем буфер символов и отправляем его в HTMLWriter
                char[] buffer = new char[4096];
                int numchars;
                while((numchars = in.read(buffer)) != -1)
                    out.write(buffer, 0, numchars);
                // Закрываем потоки
                in.close();
                out.close();
            }
            catch (IOException e) {}
        }
        /** Когда апплет заканчивает работу, закрываем созданное окно */
        public void destroy() { out.closeWindow(); }
    }
}
```

Упражнения

3-1. Напишите программу с названием `Head`, распечатывающую первые десять строк каждого заданного в командной строке файла.

3-2. Напишите симметричную первой программу с названием `Tail`, распечатывающую последние десять строк каждого заданного в командной строке файла.

3-3. Напишите программу, подсчитывающую число строк, слов и символов в заданном файле. Используйте статические методы класса `java.lang.Character` для определения того, является ли заданный символ пробелом (и, следовательно, разделителем двух слов).

3-4. Напишите программу, которая суммирует и сообщает общий объем файлов в заданном каталоге. Она должна рекурсивно обследовать все подкаталоги, суммируя и сообщая объем содержащихся в подкаталоге файлов, и подытоживать эти объемы в окончательном выводе.

3-5. Напишите программу, выдающую список всех файлов и подкаталогов, находящихся в заданном каталоге, с указанием их размеров и дат изменения. По умолчанию вывод должен быть отсортирован по имени. Если же программа вызвана с опцией `-s`, вывод должен быть отсортирован по размеру от наибольшего к наименьшему. Вызванная с опцией `-d` программа должна отсортировать список по дате от самых новых к самым старым. Для сортировки используйте метод `sort()` пакета `java.util.Collections`.

3-6. Напишите программу `Uncompress`, восстанавливающую файлы и каталоги, сжатые в примере `Compress` этой главы.

3-7. Напишите подкласс потока `OutputStream` с именем `TeeOutputStream`, который будет работать, как Т-образное соединение труб. Этот поток должен посылать свой вывод в два различных потока вывода, задаваемых при создании `TeeOutputStream`. Напишите простую тестовую программу, использующую два объекта `TeeOutputStream` для отправки текста, считанного из `System.in`, в `System.out` и в два разных тестовых файла.

3-8. Напишите простой подкласс потока `Reader` с именем `TestReader`, который все время возвращает один и тот же символ (символ, переданный конструктору). Поток никогда не доберется до конца файла. Напишите тривиальный подкласс потока `Writer` с именем `NullWriter`, который просто теряет любой посланный на него вывод. Такого рода потоки иногда полезны для тестирования и других целей. Напишите тестовую программу, читающую из потока `TestReader` и посылающую свой вывод в поток `PrintWriter`, в который вложен `NullWriter`.

Глава 4

Потоки исполнения

Поток исполнения[1] (*thread*, буквальный перевод – «нить») представляет собой составляющую программного исполнения, которая работает независимо от других потоков. Java-программы могут состоять из множества потоков, ведущих себя так, как если бы они исполнялись на независимых процессорах, даже если у главного компьютера только один процессор. Во многих языках программирования возможность организации нескольких потоков добавляется как бы вдогонку. В Java, напротив, она тесно увязана с языком и его основными пакетами:

- Интерфейс `java.lang.Runnable` определяет метод `run()`, функционирующий подобно блоку кода, исполняемого потоком. При выходе из метода поток прекращается.
- Класс `java.lang.Thread` представляет поток; он определяет методы, устанавливающие и опрашивающие свойства потока (такие как приоритет исполнения) и запускающие его.
- Можно использовать оператор и модификатор `synchronized` для написания блоков кода и, соответственно, целых методов, требую-

[1] В литературе по Java термин «thread» чаще всего переводится как «поток», чтобы подчеркнуть различие с системными процессами. Возникающая при этом переводе омонимия с «потоком» (stream) данных представляется безобидной, пока о потоках (процессах) и потоках (данных) говорится в разных главах учебника или справочника. Когда же на протяжении нескольких строк программы создается несколько потоков (процессов) и несколько потоков (ввода/вывода), после чего потоки (процессы) начинают управлять потоками (данных), русскоязычный комментарий к программе при традиционном переводе становится либо невразумительным, либо многословным. В этой книге слово «thread» переводится как «поток исполнения». – *Примеч. перев.*

щих, чтобы до начала их исполнения поток заблокировал доступ к ним других потоков. Этот механизм гарантирует, что два потока не смогут одновременно исполнять данный блок или метод, что позволяет избежать проблем, возникающих, когда несколько работающих с общими данными потоков приводят эти данные в неопределенное состояние.

- Для приостановки и возобновления потоков могут использоваться методы `wait()` и `notify()` пакета `java.lang.Object`.

Применение потоков в Java-программировании – обычное дело; невозможно ограничить их обсуждение пределами одной главы. Здесь мы начнем с простых примеров. Мы снова встретимся с потоками в главе 5 «Сетевые операции», где они оказываются очень полезными при написании программ сетевых серверов, способных одновременно отвечать на несколько клиентских запросов. Также потоки появляются в главе 11 «Графика» и в главе 15 «Апплеты», где они применяются для создания анимационных эффектов.

Основы потоков исполнения

Пример 4.1 – это простая программа, демонстрирующая, как можно определять и запускать потоки и как можно ими манипулировать. Большую часть програмы составляет метод `main()`; он запускается начальным потоком, который создается интерпретатором Java. Метод `main()` определяет два дополнительных потока, устанавливает их приоритеты и запускает их. Эти два потока определяются двумя различными способами: первый определяется путем образования подкласса класса `Thread`, тогда как второй реализует интерфейс `Runnable` и передает объект `Runnable` конструктору `Thread()`. Пример также показывет возможные применения важных методов `sleep()`, `yield()` и `join()`. Наконец, пример 4.1 демонстрирует класс `java.lang.ThreadLocal`, добавленный в Java 1.2.

Пример 4.1. ThreadDemo.java

```
package com.davidflanagan.examples.thread;

/**
 * Этот класс демонстрирует применение потоков исполнения. Метод main() - это
 * начальный метод, вызываемый интерпретатором. Он определяет и запускает еще
 * два потока исполнения, в результате одновременно будут исполняться три
 * потока. Обратите внимание на то, что этот класс является подклассом
 * класса Thread и замещает его метод run(). Этот метод
 * предоставляет тело одного из потоков, запускаемых методом main()
 **/
public class ThreadDemo extends Thread {
    /**
     * Этот метод замещает метод run() класса Thread. Он предоставляет тело
     * для этого процесса.
```

```
 **/
public void run() { for(int i = 0; i < 5; i++) compute(); }

/**
 * Метод main создает и запускает в дополнение к начальному
 * потоку исполнения, создаваемому интерпретатором для вызова
 * метода main(), еще два потока
 **/
public static void main(String[] args) {
    // Создаем первый поток, являющийся экземпляром этого класса.
    // Его телом является вышеприведенный метод run()
    ThreadDemo thread1 = new ThreadDemo();

    // Создаем второй поток исполнения, передавая объект класса Runnable
    // конструктору Thread(). Телом этого потока является метод run()
    // объекта Runnable, приведенного ниже.
    Thread thread2 = new Thread(new Runnable() {
            public void run() { for(int i = 0; i < 5; i++) compute(); }
        });

    // Устанавливаем, если они заданы, приоритеты потоков исполнения
    if (args.length >= 1) thread1.setPriority(Integer.parseInt(args[0]));
    if (args.length >= 2) thread2.setPriority(Integer.parseInt(args[1]));

    // Запускаем оба потока исполнения
    thread1.start();
    thread2.start();

    // Текущий метод main() запущен начальным потоком исполнения,
    // созданным интерпретатором Java
    // Теперь и этот процесс выполняет какую-то работу.
    for(int i = 0; i < 5; i++) compute();

    // Мы могли ожидать, что потоки исполнения завершатся, запустив
    // следующие строки. Однако здесь в них нет необходимости, поэтому
    // не станем усложнять себе жизнь.
    // try {
    //     thread1.join();
    //     thread2.join();
    // } catch (InterruptedException e) {}

    // Виртуальная машина Java завершает работу только после выхода из
    // метода main() и прекращения исполнения всех потоков
    // (это не относится к потокам-демонам - см. setDaemon()).
}

// Объекты ThreadLocal представляют значение, доступ к которому
// осуществляют методы get() и set(). Но для каждого потока
// исполнения они содержат свое значение. Этот объект отслеживает число
// вызовов метода compute() каждым из потоков исполнения.
static ThreadLocal numcalls = new ThreadLocal();

/** Это "метод-заглушка", который вызывают все наши потоки */
static synchronized void compute() {
    // Подсчитываем, сколько раз нас вызывал текущий поток исполнения
```

```
            Integer n = (Integer) numcalls.get();
            if (n == null) n = new Integer(1);
            else n = new Integer(n.intValue() + 1);
            numcalls.set(n);

            // Отображаем имя потока исполнения и число вызовов
            System.out.println(Thread.currentThread().getName() + ": " + n);

            // Выполняем длительные вычисления, имитируя нагрузку,
            // которую поток исполнения создает для процессора
            for(int i = 0, j=0; i < 1000000; i++) j += i;

            // Мы могли бы также сымитировать задержки, связанные с передачей
            // данных по сети или с вводом/выводом, заставляя поток исполнения
            // приостанавливаться на случайный промежуток времени (sleep):
            try {
                // Исполнение приостанавливается на случайное число миллисекунд
                Thread.sleep((int)(Math.random()*100+1));
            }
            catch (InterruptedException e) {}

            // Каждый поток исполнения вежливо предоставляет другим потокам
            // возможность исполняться. Важно, чтобы ресурсоемкие потоки
            // исполнения не заставляли «голодать»
            // другие потоки с таким же приоритетом.
            Thread.yield();
        }
    }
```

Потоки и группы потоков

В Java каждый поток исполнения `Thread` принадлежит какой-либо группе потоков исполнения `ThreadGroup`, и, применяя методы этой группы, можно руководить его поведением. В свою очередь, каждая группа `ThreadGroup` сама содержится в какой-то родительской группе `ThreadGroup`. То есть образуется иерархия групп потоков исполнения и потоков, содержащихся в них. Пример 4.2 демонстрирует класс `ThreadLister`, содержащий открытый (public) метод `listAllThreads()`, который отображает эту иерархию, распечатывая список всех (исполняемых под управлением интерпретатора Java в настоящий момент) потоков и их групп. Этот метод отображает имя и приоритет каждого потока исполнения, а также другую информацию о потоках и их группах. В этом примере определяется метод `main()`, создающий простой пользовательский Swing-интерфейс, которые применяется для печати его собственных потоков. На рис. 4.1 показана такая распечатка.

Метод `listAllThreads()` применяет статический метод `currentThread()` класса `Thread` для получения исполняемого в данный момент потока, а затем вызывает метод `getThreadGroup()`, чтобы найти группу, которой принадлежит этот поток. Затем этот метод использует метод `ThreadGroup.getParent()` для перемещения вверх по иерархии групп потоков, по-

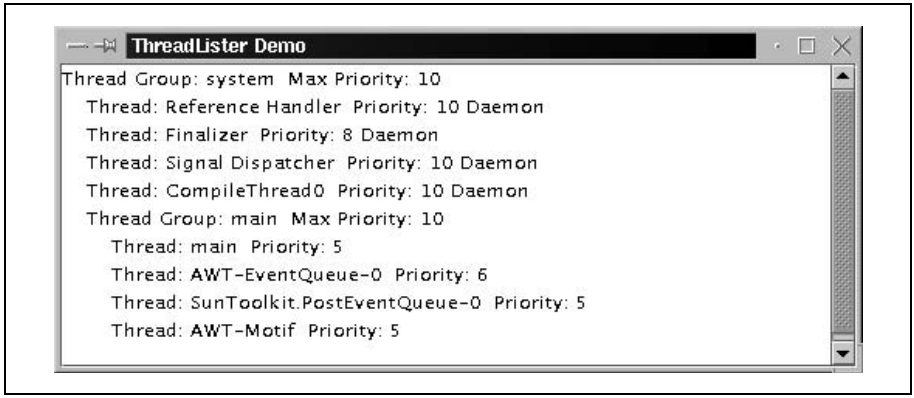


Рис. 4.1. *Потоки исполнения и группы потоков исполнения Swing-приложения*

ка не достигается корневая группа – группа потоков исполнения, содержащая все другие группы и потоки.

Затем метод `listAllThreads()` вызывает закрытый (private) метод `ThreadLister.printGroupInfo()` для отображения содержимого корневой группы потоков исполнения и далее рекурсивно отображает содержимое всех входящих в нее групп. Метод `printGroupInfo()` и вызываемый им метод `printThreadInfo()` используют различные методы классов `Thread` и `ThreadGroup` для получения информации о потоках исполнения и группах этих потоков. Обратите внимание на то, что возвращаемое методом `isDaemon()` значение сообщает, является ли поток потоком-демоном или нет. Потоки-демоны исполняются в фоновом режиме, и не предполагается, что их исполнение когда-либо прекращается. Интерпретатор Java прекращает работу, когда прекращается исполнение всех обычных потоков исполнения (не-демонов).

Класс `ThreadLister` содержит метод `main()`, так что его можно запустить как самостоятельную программу. Интереснее, разумеется, вызывать метод `listAllThreads()` из других программ; он может также оказаться полезным при диагностике проблем, связанных с потоками исполнения.

Пример 4.2. ThreadLister.java

```
package com.davidflanagan.examples.thread;
import java.io.*;
import java.awt.*;      // Классы AWT для демонстрационной программы
import javax.swing.*;  // Классы Swing GUI для демонстрационной программы

/**
 * Этот класс содержит полезный статический метод для печати всех потоков
 * исполнения и групп потоков, исполняемых на виртуальной машине.
 * Он содержит также простой метод main(), т. е. может запускаться
 * как самостоятельная программа.
 **/
```

```
public class ThreadLister {
    /** Отображаем информацию о потоке. */
    private static void printThreadInfo(PrintWriter out, Thread t,
                                        String indent) {
        if (t == null) return;
        out.println(indent + "Поток: " + t.getName() +
                    "  Приоритет: " + t.getPriority() +
                    (t.isDaemon()?" Демон":"") +
                    (t.isAlive()?"":" Не активен"));
    }

    /** Отображаем информацию о группе потоков исполнения и содержащихся
        в ней группах и потоках */
    private static void printGroupInfo(PrintWriter out, ThreadGroup g,
                                       String indent) {
        if (g == null) return;
        int num_threads = g.activeCount();
        int num_groups = g.activeGroupCount();
        Thread[] threads = new Thread[num_threads];
        ThreadGroup[] groups = new ThreadGroup[num_groups];

        g.enumerate(threads, false);
        g.enumerate(groups, false);

        out.println(indent + " Группа потоков исполнения: " + g.getName() +
                    "  Наивысший приоритет: " + g.getMaxPriority() +
                    (g.isDaemon()?"Демон":""));

        for(int i = 0; i < num_threads; i++)
            printThreadInfo(out, threads[i], indent + "    ");
        for(int i = 0; i < num_groups; i++)
            printGroupInfo(out, groups[i], indent + "    ");
    }

    /** Находим корневую группу и рекурсивно распечатываем ее содержимое */
    public static void listAllThreads(PrintWriter out) {
        ThreadGroup current_thread_group;
        ThreadGroup root_thread_group;
        ThreadGroup parent;

        // Получаем группу текущего потока исполнения
        current_thread_group = Thread.currentThread().getThreadGroup();

        // Теперь ищем корневую группу
        root_thread_group = current_thread_group;
        parent = root_thread_group.getParent();
        while(parent != null) {
            root_thread_group = parent;
            parent = parent.getParent();
        }

        // И рекурсивно ее распечатываем
        printGroupInfo(out, root_thread_group, "");
    }
```

```
    /**
     * Метод main() создает простой графический интерфейс пользователя
     * для отображения потоков исполнения. Это позволяет нам увидеть
     * потоки исполнения, занимающиеся диспетчеризацией событий
     * ("event dispatch thread"), используемые в AWT и Swing.
     **/
    public static void main(String[] args) {
        // Создаем простой интерфейс Swing GUI
        JFrame frame = new JFrame("ThreadLister Demo");
        JTextArea textarea = new JTextArea();
        frame.getContentPane().add(new JScrollPane(textarea),
                                   BorderLayout.CENTER);
        frame.setSize(500, 400);
        frame.setVisible(true);

        // Получаем строку threadlisting (распечатку потоков исполнения)
        // при помощи потока (stream) символьного вывода StringWriter
        StringWriter sout = new StringWriter();  // Для приема листинга
        PrintWriter out = new PrintWriter(sout);
        ThreadLister.listAllThreads(out);  // Выводим список потоков
                                           // исполнения в поток ввода/вывода
        out.close();
        String threadListing = sout.toString();  // Получаем листинг
                                                 // в виде строки

        // Наконец, отображаем распечатку при помощи GUI
        textarea.setText(threadListing);
    }
}
```

Взаимная блокировка

Многопоточное (multithreaded) программирование требует от программиста соблюдения ряда предосторожностей. Если, например, несколько потоков исполнения одновременно могут изменить состояние некоторого объекта, то обычно приходится применять методы с модификатором `synchronized` или оператором `synchronized`, чтобы гарантировать, что в каждый момент только один поток исполнения изменяет состояние этого объекта. Без этого один поток исполнения может совершить изменения, перечеркивающие действия другого, и объект окажется в неопределенном состоянии.

К сожалению, применение синхронизации само приводит к новым проблемам. Синхронизация потоков исполнения связана с наложением исключительной блокировки (exclusive lock). Только один поток, который в данный момент времени заблокировал синхронизированный код, может его исполнять. Если программа использует больше одной блокировки, может возникнуть ситуация, называемая «взаимной блокировкой» (deadlock). Взаимная блокировка возникает, когда два или более потоков исполнения ожидают возможности захватить ре-

сурс, заблокированный одним из ожидающих потоков. Поскольку каждый из потоков ожидает, когда ему удастся осуществить захват, ни один из них не выпускает уже захваченный ресурс, в результате чего ни один из потоков не может захватить ожидаемый ресурс. Коллизия неразрешима – все участвующие во взаимной блокировке потоки останавливаются (halt), и программа не может продолжиться.

Пример 4.3 – это простая программа, создающая ситуацию взаимной блокировки, в которой два потока пытаются захватить два разных ресурса. В этой простой программе очень легко увидеть, как возникает взаимная блокировка. Все, однако, было бы не так очевидно, если бы в ней использовались синхронизированные методы, а не простая симметричная пара операторов `synchronized`. Более сложные ситуации возникают также при большом количестве потоков исполнения и ресурсов. Вообще, проблема взаимной блокировки сложна и неприятна. Есть, однако, надежный прием, позволяющий ее избежать. Он состоит в том, чтобы все потоки исполнения всегда осуществляли все необходимые им захваты в одном и том же порядке.

Пример 4.3. Deadlock.java

```
package com.davidflanagan.examples.thread;

/**
 * Это пример того, как НЕ следует писать многопоточные программы.
 * Это программа, преднамеренно вызывающая взаимную блокировку двух потоков
 * исполнения, каждый из которых пытается захватить ресурс, захваченный
 * другим потоком. Чтобы избежать взаимной блокировки такого рода
 * при множестве захватываемых ресурсов, все потоки должны осуществлять
 * захваты всегда в одном и том же порядке.
 **/
public class Deadlock {
    public static void main(String[] args) {
        // Это два ресурса, которые потоки будут пытаться захватить
        final Object resource1 = "resource1";
        final Object resource2 = "resource2";
        // Это первый поток. Он пытается захватить
        // сначала resource1, а затем resource2
        Thread t1 = new Thread() {
                public void run() {
                    // Захват resource1
                    synchronized(resource1) {
                        System.out.println("Поток исполнения 1: захвачен
                                            resource1");

                        // Небольшая пауза, имитирующая файловый ввод/вывод или
                        // что-нибудь еще. По существу, мы просто хотим дать
                        // второму потоку шанс стартовать. Потоки исполнения
                        // и взаимная блокировка асинхронны, но мы пытаемся
                        // вызвать здесь взаимную блокировку...
                        try { Thread.sleep(50); }
                        catch (InterruptedException e) {}
```

```
                        // Теперь ждем, не удастся ли нам захватить resource2
                        synchronized(resource2) {
                            System.out.println("Поток исполнения 1: захвачен
                                                resource2");
                        }
                    }
                }
            };

        // А вот второй поток исполнения. Он пытается захватить
        // сначала resource2, а затем resource1
        Thread t2 = new Thread() {
                public void run() {
                    // Этот поток сразу же захватывает resource2
                    synchronized(resource2) {
                        System.out.println("Поток исполнения 2: захвачен
                                            resource2");

                        // Теперь, точно так же, как первый поток,
                        // он выдерживает паузу.
                        try { Thread.sleep(50); }
                        catch (InterruptedException e) {}

                        // Теперь он пытается захватить resource1. Но постойте!
                        // Поток 1 захватил resource1 и не отпустит его, пока
                        // не сможет захватить resource2. Этот же поток
                        // удерживает resource2 и не отпустит его, пока
                        // не получит resource1. Мы в тупике. Ни один поток
                        // не может продолжить исполнение, и программа зависает.
                        synchronized(resource1) {
                            System.out.println("Поток исполнения 2: захвачен
                                                resource 1");
                        }
                    }
                }
            };

        // Запускаем два наших потока исполнения. Если все будет идти по плану,
        // возникнет взаимная блокировка, и программа никогда не завершится.
        t1.start();
        t2.start();
    }
}
```

Таймеры

В Java 1.3 появляются класс `java.util.Timer` **и абстрактный класс** `java.util.TimerTask`**. Создав подкласс класса** `TimerTask` **и реализовав его метод** `run()`**, вы можете затем использовать объект** `Timer` **для создания расписания вызовов этого метода** `run()` **в заданный момент времени или в последовательные моменты, разделенные заданным временным интервалом. Один объект** `Timer` **может применяться для создания рас-**

писания вызовов нескольких объектов TimerTask. Объект Timer весьма полезен, так как его применение позволяет упростить множество программ, которые для достижения той же функциональности должны были бы создавать собственные потоки исполнения. Обратите внимание на то, что класс java.util.Timer – совсем не то же самое, что класс javax.swing.Timer из Java 1.2.

Примеры 4.4 и 4.5 – это простые реализации классов TimerTask и Timer, которые можно применять в отсутствие Java 1.3. Они реализуют те же API, что и классы Java 1.3, но содержатся в пакете com.davidflanagan.examples.thread, а не в пакете java.util. Эти реализации не обладают устойчивостью официальных версий из Java 1.3, но они полезны для простых задач и служат хорошим примером нетривиального использования потоков исполнения. Обратите особое внимание на применение методов wait() и notify() в примере 4.5. После изучения этих примеров вам, возможно, захочется сравнить их с реализациями этих классов в Java 1.3.[1]

Пример 4.4. TimerTask.java

```
package com.davidflanagan.examples.thread;

/**
 * Этот класс реализует те же API, что и java.util.TimerTask в Java 1.3.
 * Обратите внимание на то, что TimerTask может в каждый момент управляться
 * только одним таймером Timer, но при этом данная
 * реализация не настаивает на таком ограничении.
 **/
public abstract class TimerTask implements Runnable {
    boolean cancelled = false;    // Была ли отмена?
    long nextTime = -1;   // На какой момент назначено следующее исполнение?
    long period;             // С каким интервалом происходит исполнение?
    boolean fixedRate;       // Запуск исполнения с фиксированной частотой?

    protected TimerTask() {}

    /**
     * Прекращаем исполнение задачи. Возвращаем true, если задача
     * действительно исполнялась, и false - если ее исполнение
     * уже прекращено или время ее исполнения никогда не назначалось.
     **/
    public boolean cancel() {
        if (cancelled) return false;         // Уже отменена;
        cancelled = true;                    // Отменяем ее
        if (nextTime == -1) return false;    // Никогда не исполнялась;
        return true;
    }

    /**
     * На какой момент назначено очередное исполнение? Метод run() может
```

[1] Если у вас есть Java SDK™ от Sun, загляните в архив *src.jar*, поставляемый вместе с ним.

```
     * использовать этот метод, чтобы увидеть,
     * был ли он вызван в предполагаемый момент
     **/
    public long scheduledExecutionTime() { return nextTime; }

    /**
     * Подклассы должны замещать этот метод, задавая код, который
     * должен исполняться. Класс Timer будет вызывать
     * его из своего внутреннего потока исполнения.
     **/
    public abstract void run();

    // Этот метод будет использоваться классом Timer для передачи
    // классу Task сведений о расписании.
    void schedule(long nextTime, long period, boolean fixedRate) {
        this.nextTime = nextTime;
        this.period = period;
        this.fixedRate = fixedRate;
    }

    // Этот метод будет вызываться классом Timer после того,
    // как Timer вызовет метод run.
    boolean reschedule() {
        if (period == 0 || cancelled) return false; // Больше не запускать
        if (fixedRate) nextTime += period;
        else nextTime = System.currentTimeMillis() + period;
        return true;
    }
}
```

Пример 4.5. Timer.java

```
package com.davidflanagan.examples.thread;
import java.util.Date;
import java.util.TreeSet;
import java.util.Comparator;

/**
 * Этот класс является простой реализацией API java.util.Timer из Java 1.3
 **/
public class Timer {
    // В этом упорядоченном наборе хранятся задачи, за исполнение которых
    // отвечает данный Timer. Здесь для упорядочения задач по времени
    // их исполнения используется компаратор (сравнивающий метод,
    // определенный ниже).
    TreeSet tasks = new TreeSet(new TimerTaskComparator());

    // Это поток исполнения, используемый классом для исполнения
    // подконтрольных задач
    TimerThread timer;

    /** Это конструктор для создания класса Timer, не использующего
        поток-демон*/
    public Timer() { this(false); }
```

```
/** Главный конструктор: внутренний поток исполнения будет
    демоном, если это указано */
public Timer(boolean isDaemon) {
    timer = new TimerThread(isDaemon);  // TimerThread определяется ниже
    timer.start();                      // Запуск потока исполнения
}

/** Останавливаем поток исполнения timer и снимаем все
    поставленные задачи */
public void cancel() {
    synchronized(tasks) {    // Только однин поток исполнения
                             // в каждый момент!
        timer.pleaseStop();  // Устанавливаем флаг запроса
                             // на остановку потока исполнения
        tasks.clear();       // Снимаем все задачи
        tasks.notify();      // Пробуждаем поток на тот случай, если он
                             // находится в состоянии ожидания (wait()).
    }
}

/** Включаем в расписание однократное исполнение задачи
    после миллисекунд задержки */
public void schedule(TimerTask task, long delay) {
    task.schedule(System.currentTimeMillis() + delay, 0, false);
    schedule(task);
}

/** Включаем в расписание однократное исполнение задачи,
    назначенное на заданный момент времени */
public void schedule(TimerTask task, Date time) {
    task.schedule(time.getTime(), 0, false);
    schedule(task);
}

/** Включаем в расписание периодическое исполнение,
    начинающееся в заданное время */
public void schedule(TimerTask task, Date firstTime, long period) {
    task.schedule(firstTime.getTime(), period, false);
    schedule(task);
}

/** Включаем в расписание периодическое исполнение,
    стартующее после заданной задержки */
public void schedule(TimerTask task, long delay, long period) {
    task.schedule(System.currentTimeMillis() + delay, period, false);
    schedule(task);
}

/**
 * Включаем в расписание периодическое исполнение, стартующее после
 * заданной задержки. Включаем в расписание периодические исполнения
 * с фиксированной частотой (fixed-rate), когда заданный интервал
 * (миллисекунды) отсчитывается от начала предыдущего исполнения,
 * в отличие от исполнений с фиксированным интервалом (fixed-interval),
```

```
     * отсчитываемым от конца предыдущего исполнения.
     **/
    public void scheduleAtFixedRate(TimerTask task, long delay, long period)
{
        task.schedule(System.currentTimeMillis() + delay, period, true);
        schedule(task);
    }

    /** Включаем в расписание периодические исполнения,
        стартующие через заданное время */
    public void scheduleAtFixedRate(TimerTask task, Date firstTime,
                                    long period)
    {
        task.schedule(firstTime.getTime(), period, true);
        schedule(task);
    }

    // Этот внутренний метод добавляет задачу к упорядоченному набору задач
    void schedule(TimerTask task) {
        synchronized(tasks) {  // Только один поток исполнения в каждый
                               // момент может изменять tasks!
            tasks.add(task);   // Добавляем задачу к упорядоченному
                               // набору задач
            tasks.notify();    // Пробуждаем находящиеся в состоянии
                               // ожидания потоки
        }
    }

    /**
     * Этот внутренний класс применяется для упорядочивания задач
       по времени их следующего исполнения.
     **/
    static class TimerTaskComparator implements Comparator {
        public int compare(Object a, Object b) {
            TimerTask t1 = (TimerTask) a;
            TimerTask t2 = (TimerTask) b;
            long diff = t1.nextTime - t2.nextTime;
            if (diff < 0) return -1;
            else if (diff > 0) return 1;
            else return 0;
        }
        public boolean equals(Object o) { return this == o; }
    }

    /**
     * Этот внутренний класс определяет поток исполнения, запускающий каждую
     * задачу в назначенное ей по расписанию время
     **/
    class TimerThread extends Thread {
        // Этот флаг будет установлен в значение true, чтобы приказать потоку
        // остановить исполнение. Обратите внимание на то, что он объявлен
        // как volatile. Это означает, что его может асинхронно изменить
        // другой поток, так что потоки всегда должны считывать его
```

```
// действительное состояние, а не использовать кэшированное значение,
// прочитанное ранее.
volatile boolean stopped = false;

// Конструктор
public TimerThread(boolean isDaemon) { setDaemon(isDaemon); }

// Просим поток остановиться, устанавливая флаг, описанный выше
public void pleaseStop() { stopped = true; }

// Это тело потока исполнения
public void run() {
    TimerTask readyToRun = null;  // Есть ли задача, которую
                                  // пора запускать?

    // Поток исполняет цикл до тех пор, пока флаг stopped
    // не будет установлен в true.
    while(!stopped) {
        // Если есть задача, готовая к запуску, запустим ее!
        if (readyToRun != null) {
            if (readyToRun.cancelled) {  // Если произошла отмена
                                         // готовности, отменяем запуск
                readyToRun = null;
                continue;
            }
            // Запуск задачи.
            readyToRun.run();
            // Предлагаем задаче переустановить свое расписание,
            // и если она намерена снова запускаться,
            // опять включаем ее в набор задач.
            if (readyToRun.reschedule())
                schedule(readyToRun);
            // Мы запустили ее, так что теперь запускать нечего
            readyToRun = null;
            // Возвращаемся к началу цикла, чтобы посмотреть,
            // не были ли мы остановлены
            continue;
        }

        // Ставим блокировку на набор задач
        synchronized(tasks) {
            long timeout;  // сколько миллисекунд осталось
                           // до следующего исполнения?

            if (tasks.isEmpty()) {   // Если задач не осталось,
                timeout = 0;  // ждем, пока не появится (notify())
                              // следующая задача
            }
            else {
               // Если в расписании есть задачи, берем первую из них
               // Поскольку набор упорядочен, это будет следующая
               // (по порядку исполнения).
                TimerTask t = (TimerTask) tasks.first();
                // Сколько еще до следующего запуска?
```

```
                    timeout = t.nextTime - System.currentTimeMillis();
                    // Проверяем, не пора ли запускать
                    if (timeout <= 0) {
                      readyToRun = t;  // Сохраняем как готовую к запуску
                        tasks.remove(t); // Удаляем ее из набора
                       // Покидаем синхронизированный раздел на то время,
                        // пока не запустим задачу
                         continue;
                    }
                }

                // Если мы оказались здесь, значит, нет задач, готовых
                // к запуску, так что ждем времени запуска или команды
                // notify() от новой задачи, желающей встать в очередь
                try { tasks.wait(timeout); }
                catch (InterruptedException e) {}

                // Проснувшись, возвращаемся к началу цикла while
            }
        }
    }
}

/** Этот внутренний класс определяет тестовую программу */
public static class Test {
    public static void main(String[] args) {
        final TimerTask t1 = new TimerTask() { // Задача 1:
                                               // печатаем «boom»
                public void run() { System.out.println("boom"); }
            };
        final TimerTask t2 = new TimerTask() { // Задача 2:
                                               // печатаем «BOOM»
                public void run() { System.out.println("\tBOOM"); }
            };
        final TimerTask t3 = new TimerTask() { // Задача 3:
                                               // отменяем задачи
                public void run() { t1.cancel(); t2.cancel(); }
            };

        // Создаем объект timer и включаем в расписание несколько задач
        final Timer timer = new Timer();
        timer.schedule(t1, 0, 500);     // boom каждые 0,5 сек,
                                        // начав немедленно
        timer.schedule(t2, 2000, 2000); // BOOM каждые 2 сек, начав
                                        // через 2 сек
        timer.schedule(t3, 5000);       // Остановить их через 5 сек

       // Включаем в расписание заключительную задачу: начав через 5 сек,
       // провести обратный отсчет от 5, затем уничтожить (destroy) timer
       // как последний оставшийся поток исполнения, в результате чего
       // программа завершится.
        timer.scheduleAtFixedRate(new TimerTask() {
                public int times = 5;
```

```
                public void run() {
                    System.out.println(times--);
                    if (times == 0) timer.cancel();
                }
            },
                                    5000,500);
        }
    }
}
```

Упражнения

4-1. Напишите программу, получающую в командной строке список имен файлов и печатающую число строк каждого из них. Программа должна создать один поток исполнения для каждого файла и использовать эти потоки для одновременного подсчета числа строк во всех файлах. Воспользуйтесь классом `java.io.LineNumberReader`, чтобы облегчить подсчет строк. Возможно, вы захотите создать класс `LineCounter`, который расширяет класс `Thread` или реализует интерфейс `Runnable`. Создайте затем вариант программы, использующей ваш класс `LineCounter` для последовательного, а не одновременного чтения файлов. Сравните производительность многопоточной и однопоточной программ. Затраченное время оцените с помощью метода `System.currentTimeMills()`. Сравните производительность при двух, пяти и десяти файлах.

4-2. Пример 4.3 демонстрирует, как возникает взаимная блокировка при встречной попытке двух потоков получить доступ к ресурсу, занятому другим. Преобразуйте пример так, чтобы взаимная блокировка возникала при участии трех процессов, каждый из которых пытается получить доступ к коду, занятому одним из двух других.

4-3. В примере 4.3 для демонстрации взаимной блокировки применяется оператор `synchronized`. Напишите простую программу, использующую методы с модификатором `synchronized` вместо оператора `synchronized`. Взаимные блокировки такого рода тоньше, и их труднее обнаружить.

4-4. Пример 4.5 показывает реализацию API `java.util.Timer` в Java 1.3. В Java 1.2 представлен другой класс `Timer` — класс `javax.swing.Timer`. Этот класс предназначен для той же цели, но имеет другой интерфейс API. Он вызывает метод `actionPerformed()` любого числа объектов `ActionListener` один или несколько раз после заданной задержки и через заданный интервал. Прочитайте документацию по этому классу `Timer`, а затем создайте собственную реализацию этого класса. Если вы прочитали главу 10 «Графические интерфейсы пользователя (GUI)», вы знаете, что методы слушателей событий (event listeners), такие как метод `actionPerformed()`, должны

вызываться потоком диспетчеризации событий (event dispatch thread). Следовательно, ваша реализация класса `Timer` не должна прямо вызывать `actionPerformed()`, но вместо этого должна использовать метод `java.awt.EventQueue.invokeLater()` или `javax.swing.SwingUtilities.invokeLater()`, чтобы скомандовать диспетчеру событий вызвать этот метод.

4-5. Прочитав главу 5, вы, возможно, захотите вернуться к этой главе и попробовать сделать это упражнение. Класс `Server` из главы 5 — это многопоточный сетевой сервер, предоставляющий несколько служб. Он демонстрирует важные приемы сетевых операций, но также интенсивно использует потоки исполнения. В частности, эта программа создает группу `ThreadGroup`, в которой содержатся все созданные программой потоки исполнения. Преобразуйте программу `Server` так, чтобы вдобавок к этой главной группе потоков она создавала вложенные группы для каждой из предоставляемых служб. Помещайте потоки исполнения, обслуживающие каждое клиентское подключение (connection), в группу службы. Также преобразуйте программу, чтобы у каждой службы был заданный приоритет потоков исполнения и чтобы она использовала эти приоритеты при создании потоков, обслуживающих подключение к ней. Возможно, вы захотите сохранять группы потоков и приоритеты для каждой службы в полях вложенного класса `Listener`.

Глава 5

Сетевые операции

Фирма Sun Microsystems издавна придерживалась лозунга «Сеть – это компьютер». Поэтому неудивительно, что она создавала Java как язык, ориентированный на сеть. Пакет `java.net` предоставляет мощные средства работы в сети в готовом виде. Примеры из данной главы демонстрируют эти возможности на нескольких уровнях абстракции. Они показывают, как:

- применять класс `URL` для разбора (parse) различных URL-адресов и загружать сетевые ресурсы, указываемые URL-адресом;
- применять класс `URLConnection` для управления процессом загрузки сетевых ресурсов;
- писать клиентские программы, применяющие класс `Socket` для соединения через сеть;
- применять классы `Socket` и `ServerSocket` при написании серверов;
- отправлять и получать с низкими непроизводительными расходами пакеты дейтаграмм.

Загрузка содержимого URL

Пример 5.1 показывает, как при помощи класса `URL` может быть получен сетевой ресурс, расположенный по заданному URL. Этот класс служит, главным образом, для представления и анализа URL, но содержит также несколько важных методов для получения содержимого URL. Самым высокоуровневым из этих методов является `getContent()`, загружающий содержимое URL, анализирующий его и возвращающий проанализированный объект. Этот метод опирается на специально инсталлированные обработчики содержимого, фактически

осуществляющие анализ. Сам пакет Java SDK содержит обработчики содержимого для простого текста (plain text) и для нескольких распространенных форматов изображений. Если метод `getContent()` вызывается для объекта `URL`, ссылающегося на файл с простым текстом или с изображениями в формате GIF или JPEG, он возвращает объект `String` или `Image`. В большинстве случаев, когда метод `getContent()` не знает, как обрабатывется этот тип данных, он просто возвращает поток `InputStream`, чтобы вы могли считать и проанализировать данные самостоятельно.

Пример 5.1 не использует метод `getContent()`. Вместо этого он вызывает метод `openStream()`, возвращающий поток `InputStream`, из которого может быть считано содержимое URL. Этот `InputStream` по сети подключен[1] к удаленному ресурсу, заданному вышеупомянутым URL, однако класс `URL` скрывает все детали этого подключения. (В действительности подключение устанавливается классом обработчика протокола; в Java SDK есть встроенные обработчики самых распространенных протоколов, таких как *http:*, *ftp:*, *mailto:* и *file:*.)

Пример 5.1 представляет собой простую самостоятельную программу, которая получает содержимое заданного URL и сохраняет его в файле или выводит на консоль. Вы заметите, что большая часть этой программы выглядит как взятая из главы 3 «Ввод/вывод». И в самом деле, как мы увидим в этом и других примерах этой главы, почти все сетевые операции включают применение приемов ввода/вывода, изученных нами в той главе.

Пример 5.1. GetURL.java

```
package com.davidflanagan.examples.net;
import java.io.*;
import java.net.*;

/**
 * Эта простая программа использует класс URL и его метод openStream()
 * для загрузки содержимого URL и копирует его в файл или на консоль.
 **/
public class GetURL {
  public static void main(String[] args) {
    InputStream in = null;
    OutputStream out = null;
    try {
      // Проверка аргументов
      if ((args.length != 1) && (args.length != 2))
        throw new IllegalArgumentException("Неправильное число аргументов");

      // Установка потоков
      URL url = new URL(args[0]);  // Создаем объект URL
```

[1] В этой главе термин «connection» переведен как «подключение», а «socket» – как «соединение». – *Примеч. перев.*

```
      in = url.openStream();        // Открываем его поток
      if (args.length == 2)         // Получаем соответствующий поток вывода
        out = new FileOutputStream(args[1]);
      else out = System.out;

      // Теперь копируем байты из URL в поток вывода
      byte[] buffer = new byte[4096];
      int bytes_read;
      while((bytes_read = in.read(buffer)) != -1)
        out.write(buffer, 0, bytes_read);
    }
    // При возникновении исключений печатаем сообщение об ошибке
    // и о правильном формате вызова.
    catch (Exception e) {
      System.err.println(e);
      System.err.println("Формат: java GetURL <URL> [<filename>]");
    }
    finally { // Всегда закрываем за собой потоки.
      try { in.close(); out.close(); } catch (Exception e) {}
    }
  }
}
```

Класс URLConnection

Класс `URLConnection` устанавливает подключение к URL. Метод `openStream()` класса `URL`, который мы применяли в примере 5.1, – это всего лишь вспомогательный метод, создающий объект `URLConnection` и вызывающий его метод `getInputStream()`. Применение объекта `URLConnection` напрямую, минуя `openStream()`, позволяет точнее управлять процессом получения содержимого URL.

Пример 5.2 – это простая программа, показывающая, как следует применять `URLConnection` для получения информации о типе, размере, дате и других свойствах содержимого ресурса, на который указывает URL. Для адресов URL, использующих протокол HTTP, в примере также показано, как применять подкласс `HttpURLConnection` для получения дополнительной информации о подключении.

Обратите внимание на применение класса `java.util.Date` для преобразования метки времени (timestamp) (переменной типа `long`, которая содержит число миллисекунд, прошедших после полуночи 1 января 1970 г. по Гринвичу) в строку с датой и временем, понятную для человека.

Пример 5.2. GetURLInfo.java

```
package com.davidflanagan.examples.net;
import java.net.*;
import java.io.*;
import java.util.Date;
```

```
/**
 * Класс, отображающий информацию о URL.
 **/
public class GetURLInfo {
  /** Применяем класс URLConnection для получения информации о URL */
  public static void printinfo(URL url) throws IOException {
    URLConnection c = url.openConnection(); // Получаем объект URLConnection
                                            // из объекта URL
    c.connect();               // Создаем подключение к URL

    // Отображаем информацию о содержимом URL
    System.out.println(" Тип содержимого: " + c.getContentType());
    System.out.println(" Кодировка содержимого: " + c.getContentEncoding());
    System.out.println(" Размер содержимого: " + c.getContentLength());
    System.out.println(" Дата: " + new Date(c.getDate()));
    System.out.println(" Последняя модификация: " +
                       new Date(c.getLastModified()));
    System.out.println(" Срок годности: " + new Date(c.getExpiration()));

    // Если это HTTP-подключение, отображаем некоторую добавочную информацию.
    if (c instanceof HttpURLConnection) {
      HttpURLConnection h = (HttpURLConnection) c;
      System.out.println(" Метод запроса: " + h.getRequestMethod());
      System.out.println(" Сообщение ответа: " +h.getResponseMessage());
      System.out.println(" Код ответа: " + h.getResponseCode());
    }
  }

  /** Создаем объект URL, для отображения информации
      о нем вызываем printinfo(). */
  public static void main(String[] args) {
    try { printinfo(new URL(args[0])); }
    catch (Exception e) {
      System.err.println(e);
      System.err.println("Формат: java GetURLInfo <url>");
    }
  }
}
```

Отправка электронной почты при помощи URLConnection

Как я уже говорил, Java поддерживает различные протоколы URL при посредстве обработчиков протоколов, входящих в состав Java SDK. Среди них имеется и обработчик протокола *mailto:*. Пример 5.3 демонстрирует программу, использующую *mailto:* URL для отправки электронной почты. Эта программа предлагает ввести имена отправителя и получателя (или получателей) сообщения, его тему и содержание, создает соответствующий *mailto:* URL и на его основе – объект `URLConnection`. Эта программа использует методы `setDoInput()` и `setDo-`

Output(), чтобы указать, что она будет записывать данные в URLConnection. При помощи getOutputStream() она получает соответствующий поток вывода и записывает в него заголовок и тело сообщения, закрывая поток по концу тела сообщения. Программа использует системное свойство user.name и класс InetAddress, чтобы попытаться создать правильный обратный адрес отправителя электронной почты, хотя на практике это будет работать не на всех платформах.

Чтобы отправить почтовое сообщение, обработчик протокола *mailto:* должен знать, на какой компьютер (mailhost, узел связи) его следует отослать. По умолчанию он попытается отправить сообщение на тот компьютер, на котором работает сам. Некоторые компьютеры, в частности Unix-машины в сетях интранет, работают как почтовые узлы, так что это оказывается удачным решением. Другие компьютеры, такие как PC, подключенные к Интернету по телефонным линиям, должны явным образом указывать почтовый узел в командной строке. Например, если доменное имя вашего Интернет-провайдера – *isp.net*, соответствующим почтовым узлом, скорее всего, будет *mail.isp.net*. Если почтовый узел был вами указан, он сохраняется в системном свойстве mail.host, которое считывается встроенным обработчиком протокола *mailto:*.

Обратите внимание на то, что в примере 5.3 используется метод println() для вывода сообщений на консоль, однако при отправке текста через сеть используется метод print() и явно заданный признак конца строки «\n». В разных операционных системах используются различные символы конца строки, и println() применяет символ, принятый в данной системе. Между тем, большая часть сетевых служб происходит из мира Unix и использует разделитель строк, принятый в этой системе, а именно символ новой строки «\n». Здесь мы задаем его явно, поэтому наша клиентская программа будет корректно работать в Unix, Windows или системах Macintosh.

Пример 5.3. SendMail.java

```
package com.davidflanagan.examples.net;
import java.io.*;
import java.net.*;

/**
 * Эта программа отправляет e-mail при помощи mailto: URL
 **/
public class SendMail {
  public static void main(String[] args) {
    try {
      // Если пользователь задал почтовый узел, сообщаем об этом системе.
      if (args.length >= 1)
        System.getProperties().put("mail.host", args[0]);

      // Поток Reader для чтения с консоли
      BufferedReader in =
        new BufferedReader(new InputStreamReader(System.in));
```

```
        // Запрашиваем у пользователя строки from, to и subject
        System.out.print("От кого: ");
        String from = in.readLine();
        System.out.print("Кому: ");
        String to = in.readLine();
        System.out.print("Тема: ");
        String subject = in.readLine();

        // Устанавливаем сетевое подключение для отправки почты
        URL u = new URL("mailto:" + to);      // Создаем mailto: URL
        URLConnection c = u.openConnection(); // Создаем для него URLConnection
        c.setDoInput(false);          // Сообщаем, что он не будет
                                      // использоваться для ввода
        c.setDoOutput(true);          // Сообщаем, что для вывода
                                      // он использоваться будет
        System.out.println("Подключение...");  // Сообщение пользователю
        System.out.flush();           // Отображается немедленно
        c.connect();                  // Соединяемся с узлом связи
        PrintWriter out =            // Получаем поток вывода в направлении узла
          new PrintWriter(new OutputStreamWriter(c.getOutputStream()));

        // Выводим заголовки сообщения. Не позволяем пользователю сообщать
        // ложный обратный адрес
        out.print("От кого: \"" + from + "\" <" +
             System.getProperty("user.name") + "@" +
             InetAddress.getLocalHost().getHostName() + ">\n");
        out.print("Кому: " + to + "\n");
        out.print("Тема: " + subject + "\n");
        out.print("\n"); // Пустой строкой завершается список заголовков

        // Теперь просим пользователя ввести тело сообщения
        System.out.println("Введите сообщение. " +
                  "Завершите его строкой, содержащей одну точку.");
        // Построчно считываем сообщение и отправляем его в поток вывода
        String line;
        for(;;) {
          line = in.readLine();
          if ((line == null) || line.equals(".")) break;
          out.print(line + "\n");
        }

        // Закрываем (и очищаем) поток по завершении сообщения
        out.close();
        // Сообщаем пользователю, что сообщение успешно отправлено
        System.out.println("Сообщение отправлено.");
      }
      catch (Exception e) { // Обрабатываем всевозможные исключения
                            // и сообщаем об ошибках
        System.err.println(e);
        System.err.println("Формат: java SendMail [<mailhost>]");
      }
    }
}
```

Подключение к веб-серверу

Пример 5.4 показывает программу `HttpClient`, которая загружает содержимое URL с веб-сервера и выводит его в файл или на консоль. Она ведет себя так же, как программа `GetURL` из примера 5.1. Однако несмотря на подобие в поведении, реализации этих двух программ совершенно различны. В то время как `GetURL` при реализации протокола использует класс `URL` и его обработчики протоколов, `HttpClient` прямо соединяется с веб-сервером и общается с ним по протоколу HTTP. (Она применяет старую и крайне простую версию этого протокола, что позволяет сделать простой саму программу.) Вследствие этого `HttpClient` может загружать содержимое только с *http:* URL. Она не может работать с *ftp:* или другими сетевыми протоколами. Обратите внимание на то, что `HttpClient` использует класс `URL`, но только для представления URL и для его анализа, но не для подключения.

Самым интересным в примере является введение класса `Socket` (соединение), создающего сетевое соединение клиент-сервер на основе потоков (stream). Для создания сетевого подключения к другому узлу вы просто создаете объект `Socket`, задавая нужные узел и порт. Если на заданном узле работает программа (сервер), ожидающая попыток соединиться с указанным портом, конструктор `Socket()` возвращает объект `Socket`, которым можно воспользоваться для сообщения с сервером. (Если не обнаруживается сервер, в заданном узле слушающий заданный порт, или если еще что-то оказывается не в порядке – а с сетевыми соединениями многое может оказаться не в порядке – конструктор `Socket()` выдает исключение.)

Если вы не знакомы с узлами и портами, вы можете думать об узле как об отделении связи, а о порте как о почтовом ящике в нем. Подобно тому как в почтовом отделении может быть несколько почтовых ящиков, на любом узле в сети может работать одновременно несколько серверов. Для обращения к разным серверам используются разные порты. Чтобы установить подключение, необходимо правильно указать как узел, так и порт. Многие службы используют стандартные, используемые по умолчанию порты. Веб-серверы используют порт 80, POP-серверы электронной почты используют порт 110 и т. д.

После того как вы получили объект `Socket`, вы уже подключены (через сеть) к серверу. Метод `getInputStream()` соединения `Socket` возвращает поток `InputStream`, которым можно пользоваться для чтения байтов с сервера, а метод `getOutputStream()` возвращает поток `OutputStream`, которым можно пользоваться для записи байтов на сервер. Это в точности то, что делает программа `HttpClient`. Она подключается к веб-серверу, через поток вывода соединения посылает серверу HTTP запрос `GET`, а затем через поток ввода соединения читает ответ сервера. Еще раз обратите внимание на то, что запрос `GET` явным образом завершается символом новой строки «`\n`», а не зависящим от операционной системы признаком конца строки, который выдается методом `println()`.

Пример 5.4. HttpClient.java

```
package com.davidflanagan.examples.net;
import java.io.*;
import java.net.*;

/**
 * Эта программа соединяется с веб-сервером и загружает с него содержимое
 * заданного URL. Она непосредственно применяет протокол HTTP.
 **/
public class HttpClient {
  public static void main(String[] args) {
    try {
      // Проверка аргументов
      if ((args.length != 1) && (args.length != 2))
        throw new IllegalArgumentException("Неправильное число аргументов");

      // Получаем поток вывода для записи в него содержимого URL
      OutputStream to_file;
      if (args.length == 2) to_file = new FileOutputStream(args[1]);
      else to_file = System.out;

      // Теперь применяем класс URL для разбора заданного
      // пользователем URL на составные части.
      URL url = new URL(args[0]);
      String protocol = url.getProtocol();
      if (!protocol.equals("http")) // Поддерживаем ли мы протокол?
        throw new IllegalArgumentException("Должен использоваться
                                            протокол HTTP ");
      String host = url.getHost();
      int port = url.getPort();
      if (port == -1) port = 80; // Если порт не задан, используем
                                 // стандартный порт HTTP
      String filename = url.getFile();

      // Открываем сетевое подключение с заданным узлом и портом
      Socket socket = new Socket(host, port);

      // Получаем потоки ввода и вывода для соединения socket
      InputStream from_server = socket.getInputStream();
      PrintWriter to_server = new PrintWriter(socket.getOutputStream());

      // Посылаем на веб-сервер HTTP команду GET, запрашивающую файл
      // Здесь используется старая и очень простая версия протокола HTTP
      to_server.print("GET " + filename + "\n\n");
      to_server.flush(); // Отправляем немедленно!

      // Теперь читаем ответ сервера и записываем его в файл
      byte[] buffer = new byte[4096];
      int bytes_read;
      while((bytes_read = from_server.read(buffer)) != -1)
        to_file.write(buffer, 0, bytes_read);

      // Когда сервер разрывает подключение, и мы закрываем свое хозяйство
      socket.close();
```

```
            to_file.close();
        }
        catch (Exception e) {  // сообщаем о произошедших ошибках
            System.err.println(e);
            System.err.println("Формат: java HttpClient <URL> [<filename>]");
        }
    }
}
```

Простой веб-сервер

Пример 5.5 демонстрирует очень простой веб-сервер `HttpMirror`. Вместо того чтобы возвращать клиенту файл, который тот запросил, этот сервер в качестве ответа просто возвращает по адресу клиента его же запрос. Это может оказаться полезным при отладке клиентских программ и может быть интересным, если вам любопытны подробности клиентских запросов HTTP. Чтобы запустить программу, следует в качестве ее аргумента задать порт, который она будет слушать. К примеру, на узле *oxymoron.oreilly.com* я мог бы запустить сервер такой командой:

```
oxymoron% java com.davidflanagan.examples.net.HttpMirror 4444
```

Затем на своем броузере я мог бы попытаться загрузить *http://oxymoron.oreilly.com:4444/testing.html*. Сервер проигнорирует запрос файла *testing.html*, но он возвратит запрос, посланный моим броузером. Он может выглядеть примерно так:

```
GET /testing.html HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.01Gold (X11; I; Linux 2.0.18 i486)
Host: 127.0.0.1:4444
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

Главной новинкой, представленной в примере 5.5, является класс `ServerSocket`. Этот класс используется сервером или любой другой программой, которая ждет запроса на подключение от клиента. При создании объекта `ServerSocket` вы задаете порт, по которому происходит прослушивание. Для связи с клиентом вызывается метод `accept()` объекта `ServerSocket`. Этот метод блокирует поток исполнения до тех пор, пока клиент не попытается соединиться с портом, который слушает `ServerSocket`. Когда такая попытка подключения происходит, `ServerSocket` устанавливает соединение с клиентом и возвращает объект `Socket`, способный общаться с клиентом. Затем в программе должны быть вызваны методы `getInputStream()` и `getOutputStream()` объекта `Socket`, создающие потоки (stream) для чтения байтов, поступающих от клиента, и записи байтов в адрес клиента.

Обратите внимание на то, что `ServerSocket` не используется для связи между сервером и клиентом, он используется только при ожидании и

установлении подключения к клиенту. Обычно один объект ServerSocket раз за разом используется при установлении подключения к любому количеству клиентов.

Пример 5.5 совершенно прозрачен. Он создает ServerSocket и вызывает его метод accept(), как это уже было обрисовано. Когда подключается клиент, он создает потоки и посылает клиенту несколько HTTP-заголовков, сообщая ему, что запрос успешно принят и что ответом будут данные типа text/plain. Затем он читает все HTTP-заголовки клиентского запроса и возвращает их клиенту в качестве тела своего ответа. Когда он получает от клиента пустую строку, означающую конец заголовков, он закрывает подключение.

Обратите внимание на то, что тело программы HttpMirror представляет собой большой бесконечный цикл. Она соединяется с клиентом, обрабатывает запрос, а затем возвращается к началу цикла и ожидает следующего подключения клиента. Хотя этот простой сервер может превосходно служить для целей тестирования, для которых он и создавался, он содержит некий изъян – это однопоточный сервер, и он может общаться в каждый момент только с одним клиентом. Ниже в этой главе мы увидим примеры серверов, использующих много потоков исполнения и способных поддерживать подключение любого числа клиентов.

Пример 5.5. HttpMirror.java

```
package com.davidflanagan.examples.net;
import java.io.*;
import java.net.*;

/**
 * Эта программа представляет собой очень простой веб-сервер. Получив
 * HTTP-запрос, она отправляет этот запрос в качестве ответа. Это может
 * быть интересно, например, если вы хотите понять, что именно запрашивает
 * клиент или какие данные передаются при отправке формы.
 **/
public class HttpMirror {
 public static void main(String args[]) {
  try {
   // Получаем порт, по которому будет осуществляться прослушивание
   int port = Integer.parseInt(args[0]);
   // Создаем ServerSocket для прослушивания этого порта.
   ServerSocket ss = new ServerSocket(port);
   // Теперь входим в бесконечный цикл ожидания и обработки подключений.
   for(;;) {

    // Ожидаем подключения клиента. Метод переходит в состояние ожидания
    // соединения; он возвращает уже установленное соединение с клиентом
    Socket client = ss.accept();

    // Получаем потоки ввода и вывода для разговора с клиентом
    BufferedReader in = new BufferedReader(
        new InputStreamReader(client.getInputStream()));
```

```
      PrintWriter out = new PrintWriter(client.getOutputStream());

      // Отправляем наш ответ, используя протокол HTTP 1.0
      out.print("HTTP/1.0 200 \n");            // Коды версии и статуса
      out.print("Content-Type: text/plain\n"); // Тип данных
      out.print("\n");                         // Заголовки закончились

      // Теперь считываем HTTP-запрос клиента и возвращаем его клиенту
      // в виде части тела нашего ответа. Клиент не отсоединяется, поэтому мы
      // никогда не получаем EOF. Он, однако, посылает после заголовков
      // пустую строку. Поэтому, когда мы видим пустую строку, мы перестаем
      // читать. Это значит, в частности, что мы не будем возвращать содержимое
      // запросов POST, к примеру. Заметим, что метод readLine()
      // работает с признаками конца строки, принятыми в Unix, Windows и Mac.
      String line;
      while((line = in.readLine()) != null) {
       if (line.length() == 0) break;
       out.print(line + "\n");
      }

      // Закрываем объект Socket, разрывая подключение к клиенту, и
      // закрываем потоки ввода и вывода
      out.close();    // Проталкиваем и закрываем поток вывода
      in.close();     // Закрываем поток ввода
      client.close(); // Закрываем сам объект Socket
     } // И возвращаемся к началу цикла, где будем ожидать нового подключения
    }
    // Если что-то не в порядке, печатаем сообщение об ошибке
    catch (Exception e) {
     System.err.println(e);
     System.err.println("Формат: java HttpMirror <port>");
    }
   }
  }
```

Прокси-сервер

Пример 5.6 демонстрирует еще один сетевой сервер: простой однопоточный прокси-сервер. Прокси-сервер действует как представитель (proxy) реального сервера. Когда клиент соединяется с прокси-сервером, последний перенаправляет его запросы в адрес реального сервера, а затем возвращает клиенту ответы сервера. Для клиента прокси-сервер выглядит как настоящий сервер. Для настоящего сервера прокси выглядит как клиент.

Зачем вообще возиться с прокси? Почему бы клиенту не соединяться с реальным сервером напрямую? Во-первых (и это главное), прокси-серверы применяются по соображениям безопасности. У таких серверов есть также интересные приложения, связанные с фильтрацией; изощренные прокси-серверы могут, например, вырезать с загружаемых страниц рекламу. Есть и еще одна причина применения прокси-серве-

ров, возникающая при использовании сетевых возможностей Java, связанных с апплетами, как это поясняет следующий сценарий.

Допустим, я разработал новый замечательный сервер, который работает на моем компьютере *oxymoron.oreilly.com*. Затем я пишу апплет-клиент для моей службы и публикую этот апплет в сети, размещая его на сервере *www.oreilly.com*. Тут и возникает проблема. Ограничения, налагаемые на апплеты по соображениям безопасности, позволяют апплету устанавливать сетевые подключения только с одним узлом – с узлом, с которого он был получен. Итак, мой апплет-клиент, полученный с *www.oreilly.com*, не может сообщаться с моим невероятно полезным сервером на *oxymoron.oreilly.com*. Что же делать? Системный администратор не разрешит установить мой сервер на *www.oreilly.com*, а мне не нужны издержки, связанные с запуском Apache или другого веб-сервера на моей рабочей станции *oxymoron*. И вот вместо этого я запускаю на *oxymoron* простой прокси-сервер, как в примере 5.6. Я настраиваю его так, чтобы он ожидал (listen) запросов на подключение по порту 4444 и действовал как прокси-сервер по отношению к веб-серверу, работающему по порту 80, на *www.oreilly.com*. Я публикую мой апплет на *http://www.oreilly.com/staff/david/nifty.html*, но предлагаю загружать его с *http://oxymoron.oreilly.com:4444/staff/david/nifty.html*.[1] Это решение проблемы. Как только апплет кому-нибудь понадобится, он оказывается загруженным с *oxymoron.oreilly.com* и, таким образом, может связаться с моей замечательной службой.

В примере 5.6 нет ничего нового. Интересен он тем, что соединяет черты клиента и сервера в одной программе. Изучая этот код, помните, что прокси-сервер посредник подключения между сервером и клиентом. Он действует как сервер по отношению к клиенту и как клиент по отношению к серверу. `SimpleProxyServer` – это однопоточный сервер; в каждый момент он может обрабатывать только одно подключение. Тем не менее вы заметите, что он все же использует поток исполнения (реализованный в безымянном внутреннем классе) для того, чтобы прокси мог одновременно передавать данные и от клиента серверу, и от сервера – клиенту. В этом примере главный поток исполнения считывает байты с сервера и посылает их клиенту. Отдельный поток исполнения считывает байты от клиента и посылает их на сервер. В Java 1.3 и более ранних версиях при необходимости управлять несколькими потоками ввода/вывода приходится использовать несколько потоков исполнения: Java не поддерживает неблокирующий ввод/вывод, так же как не определяет никакого способа одновременно заблокировать несколько потоков. Использование же по отдельному потоку исполнения на каждый поток на загруженных серверах может вызвать проблемы с масштабированием, так что это ограничение, возможно, будет снято в Java 1.4.

[1] Это вымышленный пример; упомянутые URL ничего не значат!

Пример 5.6. SimpleProxyServer.java

```
package com.davidflanagan.examples.net;
import java.io.*;
import java.net.*;

/**
 * Этот класс реализует простой однопоточный прокси-сервер.
 **/
public class SimpleProxyServer {
 /** Метод main() анализирует аргументы и передает их методу runServer */
 public static void main(String[] args) throws IOException {
  try {
   // Проверяем число аргументов
   if (args.length != 3)
    throw new IllegalArgumentException("Неправильное число аргументов.");

   // Получаем заданные в командной строке аргументы: узел и порт, которые мы
   // представляем, а также локальный порт, по которому мы ожидаем подключений.
   String host = args[0];
   int remoteport = Integer.parseInt(args[1]);
   int localport = Integer.parseInt(args[2]);
   // Печатаем стартовое сообщение
   System.out.println("Запускаем прокси для " + host + ":" +
        remoteport + " по порту " + localport);
   // И запускаем сервер
   runServer(host, remoteport, localport); // Исполнение никогда не прекращается
  }
  catch (Exception e) {
   System.err.println(e);
   System.err.println("Формат: java SimpleProxyServer " +
        "<host> <remoteport> <localport>");
  }
 }

 /**
  * Этот метод запускает однопоточный прокси-сервер для host:remoteport
  * на заданном локальном порте. Он никогда не прекращает работу.
  **/
 public static void runServer(String host, int remoteport, int localport)
  throws IOException {
  // Создаем ServerSocket, ожидающий подключений к нему
  ServerSocket ss = new ServerSocket(localport);

  // Создаем буферы для передачи от клиента к серверу и от сервера к клиенту.
  // Один из них объявлен как final, поэтому он может использоваться
  // в приведенном ниже безымянном классе. Обратите внимание на предположения
  // относительно объема передачи в каждом направлении.
  final byte[] request = new byte[1024];
  byte[] reply = new byte[4096];

  // Это сервер, который никогда не прекращает работу, так что входим
  // в бесконечный цикл.
  while(true) {
```

```
// Переменные, в которых будут храниться объекты Socket
// для соединения с клиентом и с сервером.
Socket client = null, server = null;
try {
 // Ожидаем подключения к локальному порту,
 client = ss.accept();

 // Получаем потоки для связи с клиентом. Определяем их как final, так что
 // ими можно будет пользоваться в приведенном ниже безымянном классе.
 final InputStream from_client = client.getInputStream();
 final OutputStream to_client = client.getOutputStream();

 // Подключаемся к реальному серверу.
 // Если подключиться не удается, посылаем клиенту сообщение об ошибке,
 // отключаемся от него и продолжаем ожидать новых подключений.
 try { server = new Socket(host, remoteport); }
 catch (IOException e) {
  PrintWriter out = new PrintWriter(to_client);
  out.print("Прокси-сервер не смог соединиться с " + host + ":"+
     remoteport + ":\n" + e + "\n");
  out.flush();
  client.close();
  continue;
 }

 // Получаем потоки для общения с сервером.
 final InputStream from_server = server.getInputStream();
 final OutputStream to_server = server.getOutputStream();

 // Создаем поток исполнения для чтения клиентских запросов и передачи их
 // серверу. Нам приходится использовать отдельный процесс, так как
 // запросы и ответы могут поступать асинхронно.
 Thread t = new Thread() {
  public void run() {
   int bytes_read;
   try {
    while((bytes_read=from_client.read(request))!=-1) {
     to_server.write(request, 0, bytes_read);
     to_server.flush();
    }
   }
   catch (IOException e) {}

   // Клиент закрыл подключение к нам, так что закрываем
   // наше подключение к серверу. Это повлечет за собой также
   // выход из цикла от сервера к клиенту в главном потоке исполнения.
   try {to_server.close();} catch (IOException e) {}
  }
 };

 // Запускаем поток исполнения запросов от клиента к серверу
 t.start();

 // В то же время в главном потоке исполнения считываем ответы сервера
```

```
        // и передаем их клиенту. Это будет делаться параллельно с только что
        // созданным потоком исполнения запросов от клиента к серверу.
        int bytes_read;
        try {
         while((bytes_read = from_server.read(reply)) != -1) {
          to_client.write(reply, 0, bytes_read);
          to_client.flush();
         }
        }
        catch(IOException e) {}

        // Сервер закрыл подключение к нам, так что и мы закрываем свое
        // подключение к нашему клиенту. Это повлечет
        // за собой завершение исполнения другого потока.
        to_client.close();
       }
       catch (IOException e) { System.err.println(e); }
       finally { // Как бы то ни было, закрываем соединения.
        try {
         if (server != null) server.close();
         if (client != null) client.close();
        }
        catch(IOException e) {}
       }
      }
     }
    }
```

Сетевые операции с апплетами

Как я уже описал, на сетевую активность ненадежных (untrusted) апплетов накладываются строгие ограничения. Им разрешается устанавливать подключение только к тому узлу, с которого они были загружены. Никакая другая деятельность в сети им не позволена. Все это, тем не менее, оставляет открытыми некоторые интересные возможности, и пример 5.7 иллюстрирует одну из них. Мы будем обсуждать апплеты и графические интерфейсы пользователя, отображаемые с их помощью, в главе 15 «Апплеты» и в главе 10 «Графические интерфейсы пользователя (GUI)». Пока вы не прочли эти главы, возможно, вам в этом примере будет не все понятно. Тем не менее данный пример содержит интересную реализацию сетевых операций, особенно в методе `run()`.

finger – это программа и сетевая служба, обычно запускаемая почти на всех Unix-машинах. Она позволяет клиенту осуществлять сетевые подключения и получать список пользователей, зарегистрированных в другой системе. Она может также позволить клиенту получать и более подробную информацию (такую как номера телефонов) об индивидуальных пользователях. В наши дни в связи с возросшим значением вопросов безопасности многие Unix-машины не предоставляют эту услугу.

Показанный в примере 5.7 апплет является клиентом службы *finger*. Предположим, что я старый Unix-хакер, и думаю, что служба *finger* – это замечательная вещь. Я запускаю сервер *finger* на своей рабочей станции и рекомендую своим друзьям пользоваться им, чтобы знать, когда я вхожу в систему. К несчастью, мои друзья работают на PC и поэтому не имеют доступа к старомодной текстовой клиентской программе *finger*, которая есть на всех Unix-машинах. Апплет `Who` из примера 5.7 решает эту проблему. Он подключается к соответствующему порту на моем узле и сообщает им, зарегистрирован я в системе или нет. Разумеется, чтобы это сработало, апплет должен быть получен с моего узла. Я мог бы воспользоваться программой `SimpleProxyServer` из примера 5.6, чтобы превратить свой компьютер в прокси-веб-сервер. (Или применить `SimpleProxyServer` для установки прокси-*finger*-сервера для своего узла на узле, где запущен веб-сервер.)

Если в вашей системе сам сервер *finger* не работает, применить этот апплет будет не так просто. Вы можете поискать где-то в Сети узел, предоставляющий службу *finger*. (Я, например, использовал *rtfm.mit.edu*.) Затем вы можете применить `SimpleProxyServer`, чтобы запустить локальный прокси для этого *finger*-сервера, и запустить апплет локально на своей машине. Вам, однако, придется слегка видоизменить апплет, чтобы он соединялся с требуемым прокси-портом, а не с зарезервированным портом *finger* (порт 79).

Пример 5.7. Who.java

```
package com.davidflanagan.examples.net;
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;

/**
 * Этот апплет соединяется с finger-сервером на узле, с которого он получен,
 * чтобы узнать, кто в настоящий момент зарегистрирован в системе.
 * Так как это ненадежный апплет, он может соединяться только с тем узлом,
 * с которого он был загружен. Поскольку веб-серверы сами редко запускают
 * finger-серверы, этот апплет будет часто использоваться вместе
 * с прокси-сервером, чтобы обслуживаться с узла, на котором
 * finger-сервер есть.
 **/
public class Who extends Applet implements ActionListener, Runnable {
 Button who; // Кнопка в апплете

 /**
  * Метод init() просто создает кнопку, которая будет использоваться в апплете.
  * Когда пользователь щелкнет на кнопке, мы увидим, кто зарегистрирован.
  **/
 public void init() {
  who = new Button("Who?");
```

```
 who.setFont(new Font("SansSerif", Font.PLAIN, 14));
 who.addActionListener(this);
 this.add(who);
}

/**
 * После нажатия кнопки запускаем процесс, который подключится
 * к finger-серверу и покажет список зарегистрированных пользователей
 **/
public void actionPerformed(ActionEvent e) { new Thread(this).start(); }

/**
 * Это метод, осуществляющий сетевые операции и отображающий результаты.
 * Он реализован как отдельный поток исполнения, поскольку на его завершение
 * может понадобиться некоторое время, а методы апплета должны
 * выполняться быстро.
 **/
public void run() {
 // Отключаем кнопку, так что за один раз нельзя будет отправить
 // несколько запросов...
 who.setEnabled(false);

 // Создаем окно для отображения в нем вывода
 Frame f = new Frame("Список зарегистрированных пользователей: Подключение...");
 f.addWindowListener(new WindowAdapter() {
   public void windowClosing(WindowEvent e) {
    ((Frame)e.getSource()).dispose();
   }
  });
 TextArea t = new TextArea(10, 80);
 t.setFont(new Font("MonoSpaced", Font.PLAIN, 10));
 f.add(t, "Center");
 f.pack();
 f.show();

 // Смотрим, кто зарегистрирован
 Socket s = null;
 PrintWriter out = null;
 BufferedReader in = null;
 try {
  // Соединяемся с портом 79 (стандартный finger-порт) на узле,
  // с которого апплет был загружен.
  String hostname = this.getCodeBase().getHost();
  s = new Socket(hostname, 79);
  // Создаем потоки
  out = new PrintWriter(new OutputStreamWriter(s.getOutputStream()));
  in = new BufferedReader(new InputStreamReader(s.getInputStream()));

  // Отправляем на finger-сервер пустую строку, говоря тем самым,
  // что нам интересен список зарегистрированных пользователей,
  // а не информация об отдельном пользователе.
  out.print("\n");
  out.flush(); // Отправляем немедленно!
```

```
    // Теперь читаем ответ сервера и отображаем его в текстовой области.
    // Сервер должен посылать строки, оканчивающиеся символом \n.
    // Метод readLine() должен распознавать конец строки, даже если
    // он работает на Mac, где строки завершаются символом \r
    String line;
    while((line = in.readLine()) != null) {
     t.append(line);
     t.append("\n");
    }
    // Обновляем заголовок окна, показывая, что мы закончили
    f.setTitle("Кто зарегистрирован на: " + hostname);
   }
   // Если что-то оказывается не в порядке, просто отображаем сообщение об ошибке
   catch (IOException e) {
    t.append(e.toString());
    f.setTitle("Список зарегистрированных пользователей: Ошибка");
   }
   // И в конце не забываем закрыть потоки!
   finally {
    try { in.close(); out.close(); s.close(); } catch(Exception e) {}
   }

   // И снова делаем кнопку активной
   who.setEnabled(true);
  }
 }
```

Универсальный клиент

Класс `HttpClient` из примера 5.4 был специализированным клиентом. Пример 5.8 определяет класс `GenericClient`, способный выполнять функции клиента для различных текстовых служб. Будучи запущенной, эта программа соединяется с узлом и портом, заданными в командной строке. После этого она просто пересылает серверу набранный вами текст, а текст ответа сервера выводит на консоль.

Класс `GenericClient` может использоваться для загрузки файлов с веб-сервера путем отправки простой команды `GET` в соответствии с протоколом HTTP, как это делалось в программе `HttpClient`. Для больших файлов, однако, это не очень полезно, так как ответ сервера будет слишком быстро пробегать по экрану. `GenericClient` будет полезней при использовании интерактивных текстовых протоколов. Таким протоколом является Post Office Protocol (POP). Программой `GenericClient` можно воспользоваться для предварительного просмотра ожидаемой электронной почты на вашем ISP (или где-нибудь еще). Диалог с POP-сервером при посредстве `GenericClient` может выглядеть приблизительно так (жирным выделены строки, которые вводит пользователь):

```
oxymoron% java com.davidflanagan.examples.net.GenericClient mail.isp.net 110
Connected to mail.isp.net/208.99.99.251:110
+OK QUALCOMM Pop server derived from UCB (version 2.1.4-R3) at mail.isp.net
```

```
starting.
user djf
+OK Password required for djf.
pass notrealpassword
+OK djf has 3 message(s) (2861 octets).
retr 3
+OK 363 octets
Received: from obsidian.oreilly.com (obsidian.oreilly.com [207.144.66.251])
  by mail.isp.net (8.8.5/8.8.5) with SMTP id RAA11654
  for djf@isp.net; Wed, 21 Jun 2999 17:01:50 -0400 (EDT)
Date: Wed, 25 Jun 1997 17:01:50 -0400 (EDT)
Message-Id: &lt;199706252101.RAA11654@mail.isp.net&gt;
From: "Paula Ferguson" &lt;pf@oreilly.com&gt;
To: djf@isp.net
Subject: schedule!

Aren't you done with that book yet?
.
dele 3
+OK Message 3 has been deleted.
quit
+OK Pop server at mail.isp.net signing off.
Connection closed by server.
oxymoron%
```

Класс `GenericClient` имеет структуру, весьма схожую с классом `SimpleProxyServer` из примера 5.6. Подобно `SimpleProxyServer`, `GenericClient` использует второй безымянный поток исполнения. Он передает данные от сервера клиенту параллельно с основным потоком исполнения, передающим данные от клиента серверу. Обратите внимание на то, что поток исполнения, передающий данные от сервера клиенту, преобразует получаемый от сервера символ конца строки «`\n`» в локальный символ конца строки, каким бы тот ни был. За счет применения двух потоков исполнения пользовательский ввод и вывод сервера могут происходить асинхронно, что и делается при использовании некоторых протоколов. Единственная тонкость, связанная с `GenericClient`, состоит в том, что у этих двух потоков исполнени должны быть разные приоритеты, поскольку в некоторых реализациях Java поток исполнения не может писать на консоль в то время, когда другой поток с таким же[1] приоритетом заблокирован в ожидании ввода с консоли.

Пример 5.8. GenericClient.java

```
package com.davidflanagan.examples.net;
import java.io.*;
import java.net.*;

/**
 * Эта программа соединяется с сервером на заданном узле и порте.
```

[1] Или более высоким. – *Примеч. науч. ред.*

```
 * Она считывает текст с консоли и отправляет его серверу.
 * Она считывает текст от сервера и выводит его на консоль.
 **/
public class GenericClient {
 public static void main(String[] args) throws IOException {
 try {
  // Проверяем число аргументов
  if (args.length != 2)
  throw new IllegalArgumentException("Неправильное число аргументов");

  // Анализируем аргументы, заданные для узла и порта
  String host = args[0];
  int port = Integer.parseInt(args[1]);

  // Соединяемся с заданными узлом и портом
  Socket s = new Socket(host, port);

  // Создаем потоки для связи с сервером. Поток from_server объявлен
  // как final ввиду его использования приведенным ниже внутренним классом
  final Reader from_server=new InputStreamReader(s.getInputStream());
  PrintWriter to_server = new PrintWriter(s.getOutputStream());

  // Создаем потоки для связи с консолью. Поток to_user объявлен
  // как final ввиду его использования приведенным ниже безымянным классом
  BufferedReader from_user =
  new BufferedReader(new InputStreamReader(System.in));
  // Параметр true передается для осуществления автоматического
  // проталкивания в методе println()
  final PrintWriter to_user = new PrintWriter(System.out, true);

  // Сообщаем пользователю, что подключение установлено
  to_user.println("Установлено подключение к " + s.getInetAddress() +
    ":" + s.getPort());

  // Создаем поток исполнения, получающий вывод с сервера и отображающий его
  // для пользователя. Поскольку для этого используется отдельный поток
  // исполнения, вывод можно получать асинхронно
  Thread t = new Thread() {
  public void run() {
   char[] buffer = new char[1024];
   int chars_read;
   try {
   // Считываем символы, пока поток ввода не закроется
   while((chars_read = from_server.read(buffer)) != -1) {
    // Проходим цикл по массиву символов и печатаем их,
    // преобразуя все символы \n в локальный признак конца строки.
    // Можно было бы сделать и поэффективней, но это, вероятно, будет
    // происходить быстрее, чем передача по сети, так что нам этого хватит
    for(int i = 0; i < chars_read; i++) {
    if (buffer[i] == '\n') to_user.println();
    else to_user.print(buffer[i]);
    }
    to_user.flush();
   }
```

```
    }
    catch (IOException e) { to_user.println(e); }

    // Когда сервер закрывает подключение, вышеприведенный цикл
    // заканчивается. Сообщаем пользователю о произошедшем и вызываем
    // System.exit(), в результате чего наряду с этим потоком исполнения
    // прекратит работу и основной поток.
    to_user.println("Сервер закрыл подключение.");
    System.exit(0);
   }
   };

   // Приоритет потока исполнения, передающего данные от сервера
   // к пользователю, делаем на единицу больше, чем у основного потока.
   // Это как будто необязательно, но в некоторых операционных системах
   // вывод на консоль не отобразится, пока процесс с таким же
   // приоритетом заблокирован в ожидании ввода с консоли.
   t.setPriority(Thread.currentThread().getPriority() + 1);

   // Теперь запускаем этот поток исполнения
   t.start();

   // Параллельно с ним считываем пользовательский ввод
   // и отправляем его серверу.
   String line;
   while((line = from_user.readLine()) != null) {
   to_server.print(line + "\n");
   to_server.flush();
   }

   // Если пользователь нажимает Ctrl-D (Unix) или Ctrl-Z (Windows)
   // для завершения своего ввода, мы отправляем EOF, и происходит выход
   // из цикла. Когда это происходит, мы останавливаем дополнительный
   // (server-to-user) поток исполнения и закрываем соединение (socket).

   s.close();
   to_user.println("Подключение закрыто клиентом.");
   System.exit(0);
  }
  // Если что-то не в порядке, печатаем сообщение об ошибке
  catch (Exception e) {
   System.err.println(e);
   System.err.println("Формат: java GenericClient <hostname> <port>");
  }
  }
}
```

Универсальный многопоточный сервер

Пример 5.9 – длинный и весьма сложный пример. Класс `Server`, который в нем определяется, – это многопоточный сервер, предоставляющий услуги, определяемые реализациями вложенного интерфейса

Server.Service. Он может предоставлять несколько служб (определяемых несколькими объектами Service) на нескольких портах, и способен динамически загружать классы Service и создавать их экземпляры и добавлять (и удалять) новые службы в процессе работы. Он регистрирует свои действия в заданном потоке (stream) и ограничивает количество одновременных подключений заданным числом.

Класс Server использует несколько внутренних классов. Класс Server.Listener является потоком исполнения, ожидающим подключений по заданному порту. На каждую службу, предоставляемую сервером Server, создается свой объект Listener. Класс Server.ConnectionManager управляет списком имеющихся на данный момент подключений ко всем службам. Все службы совместно используют один объект ConnectionManager. Когда Listener получает подключение к клиенту, он передает его менеджеру ConnectionManager, который отклоняет это подключение, если предельное число подключений уже достигнуто. Если ConnectionManager не отказывается от клиента, он создает объект Server.Connection для обработки подключения. Класс Connection является подклассом класса Thread, поэтому каждая служба может обрабатывать несколько подключений одновременно, что и делает наш сервер многопоточным. Каждый объект Connection передается объекту Service и вызывает его метод serve(), который фактически и предоставляет запрашиваемую услугу.

Интерфейс Service является вложенным (nested) членом класса Server; Server включает в себя множество реализаций этого класса. Многие из этих реализаций являются простейшими демонстрационными службами. Однако класс Control – это нетривиальная реализация Service. Эта служба предоставляет защищенный паролем доступ к серверу во время исполнения, позволяя удаленному администратору добавлять и удалять службы, проверять статус сервера и изменять текущее ограничение на число подключений.

Наконец, метод main() класса Server является самостоятельной программой, создающей и запускающей Server. Задавая в командной строке аргумент -control, можно предписать программе создать экземпляр службы Control, позволяющей администрировать сервер во время исполнения. Другие аргументы этой программы задают имена классов Service, которые следует запустить, и порты, которыми они будут пользоваться. Например, можно было бы запустить сервер следующей командой:

```
% java com.davidflanagan.examples.net.Server -control secret 3000 \
  com.davidflanagan.examples.net.Server\$Time 3001 \
  com.davidflanagan.examples.net.Server\$Reverse 3002
```

Эта команда запускает службу Control на порте 3000 с паролем «secret», службу Server.Time на порте 3001 и службу Server.Reverse на порте 3002. Запустив программу сервера, вы могли бы воспользоваться программой GenericClient (см. пример 5.8) для подключения к каждой предос-

тавляемой сервером службе. Применение службы Control, конечно, интереснее всего, ее можно использовать для добавления (и удаления) других служб.

Лучшим способом понять класс Server и его внутренние классы и интерфейсы будет погрузиться в него и изучить его коды. Они хорошо прокомментированы. Я бы рекомендовал просмотреть программу целиком, обращая внимание на комментарии, а затем вернуться к подробному изучению каждого класса.

Пример 5.9. Server.java

```
package com.davidflanagan.examples.net;
import java.io.*;
import java.net.*;
import java.util.*;

/**
 * Этот класс представляет собой универсальный шаблон настраиваемого
 * многопоточного сервера. Он ожидает подключений по любому числу
 * заданных портов и, получив подключение к порту, передает потоки ввода
 * и вывода заданному объекту Service, осуществляющему реальное обслуживание.
 * Он может ограничивать число одновременных подключений и регистрировать
 * свои действия в заданном потоке.
 **/
public class Server {
 /**
 * Метод main() для запуска сервера как самостоятельной программы.
 * Аргументы программы, задаваемые в командной строке, должны образовывать
 * пары, состоящие из названия службы и номера порта. Для каждой пары
 * программа динамически загружает названный класс Service, создает его
 * экземпляр и приказывает серверу предоставить эту службу по заданному
 * порту. Специальный аргумент -control, за которым должны следовать
 * пароль и порт, запускает специальную управляющую службу сервера,
 * работающую на заданном порте, защищенном заданным паролем.
 **/
 public static void main(String[] args) {
 try {
  if (args.length < 2) // Проверяем число аргументов
  throw new IllegalArgumentException("Должен быть указан сервис");

  // Создаем объект Server, использующий стандартный вывод в качестве
  // регистрационного журнала и ограничивающий число
  // одновременных подключений десятью.
  Server s = new Server(System.out, 10);

  // Анализ списка аргументов
  int i = 0;
  while(i < args.length) {
  if (args[i].equals("-control")) { // Обработка аргумента -control
   i++;
   String password = args[i++];
   int port = Integer.parseInt(args[i++]);
```

```
  // добавляем управляющую службу
  s.addService(new Control(s, password), port);
 }
 else {
  // В противном случае запускаем названную службу по заданному порту.
  // Динамически загружаем класс Service и создаем его экземпляр.
  String serviceName = args[i++];
  Class serviceClass = Class.forName(serviceName);
  Service service = (Service)serviceClass.newInstance();
  int port = Integer.parseInt(args[i++]);
  s.addService(service, port);
 }
 }
}
catch (Exception e) { // Отображаем сообщение, если что-то не в порядке.
 System.err.println("Сервер: " + e);
 System.err.println("Формат: java Server " +
   "[-control <password> <port>] " +
   "[<servicename> <port> ... ]");
 System.exit(1);
}
}

// Параметры состояния сервера
Map services;   // Хеш-таблица, связывающая порты с объектами Listener
Set connections;  // Набор текущих подключений
int maxConnections;  // Лимит одновременных подключений
ThreadGroup threadGroup; // Группа всех наших потоков исполнения
PrintWriter logStream;  // Сюда мы направляем наш регистрационный вывод

/**
* Это конструктор сервера Server(). Ему должны передаваться поток (stream),
* в который направляется регистрационный вывод (возможно, null),
* и максимальное число одновременных подключений.
**/
public Server(OutputStream logStream, int maxConnections) {
setLogStream(logStream);
log("Сервер запущен");
threadGroup = new ThreadGroup(Server.class.getName());
this.maxConnections = maxConnections;
services = new HashMap();
connections = new HashSet(maxConnections);
}

/**
* Открытый (public) метод, устанавливающий текущий регистрационный поток.
* Аргументу null соответствует отключение регистрации.
**/
public synchronized void setLogStream(OutputStream out) {
if (out != null) logStream = new PrintWriter(out);
else logStream = null;
}
```

```
/** Записываем заданную строку в регистрационный журнал */
protected synchronized void log(String s) {
if (logStream != null) {
 logStream.println("[" + new Date() + "] " + s);
 logStream.flush();
}
}
/** Записываем заданный объект в регистрационный журнал */
protected void log(Object o) { log(o.toString()); }

/**
* Этот метод заставляет сервер открыть новую службу.
* Он запускает заданный объект Service на заданном порте.
**/
public synchronized void addService(Service service, int port)
throws IOException
{
Integer key = new Integer(port); // ключ хеш-таблицы
// Проверяем, не занят ли этот порт какой-либо службой
if (services.get(key) != null)
 throw new IllegalArgumentException("Порт " + port +
     " уже используется.");
// Создаем объект Listener, который будет ожидать подключений к этому порту
Listener listener = new Listener(threadGroup, port, service);
// Сохраняем его в хеш-таблице
services.put(key, listener);
// Регистрируем событие
log("Запуск службы " + service.getClass().getName() +
 " по порту " + port);
// Запускаем listener.
listener.start();
}

/**
* Этот метод заставляет сервер закрыть службу по заданному порту.
* Он не закрывает существующие подключения к этой службе, а просто
* приказывает серверу прекратить принимать новые подключения.
**/
public synchronized void removeService(int port) {
Integer key = new Integer(port); // Ключ хеш-таблицы
// Ищем в хеш-таблице объект Listener, соответствующий заданному порту
final Listener listener = (Listener) services.get(key);
if (listener == null) return;
// Просим listener остановиться
listener.pleaseStop();
// Удаляем его из хеш-таблицы
services.remove(key);
// И регистрируем событие.
log("Останов службы " + listener.service.getClass().getName() +
 " по порту " + port);
}
```

```
/**
* Этот вложенный подкласс класса Thread «слушает сеть». Он ожидает
* попыток подключиться к заданному порту (с помощью ServerSocket), и когда
* получает запрос на подключение, вызывает метод сервера addConnection(),
* чтобы принять (или отклонить) подключение. Для каждой службы Service,
* предоставляемой сервером Server, есть один объект Listener.
**/
public class Listener extends Thread {
ServerSocket listen_socket;    // Объект ServerSocket, ожидающий подключений
int port;                      // Прослушиваемый порт
Service service;               // Служба по этому порту
volatile boolean stop = false; // Признак команды остановки

/**
 * Конструктор Listener создает для себя поток исполнения в составе заданной
 * группы. Он создает объект ServerSocket, ожидающий подключений
 * по заданному порту. Он настраивает ServerSocket так, чтобы его
 * можно было прервать, за счет чего служба может быть удалена с сервера.
 **/
public Listener(ThreadGroup group, int port, Service service)
 throws IOException
{
 super(group, "Listener:" + port);
 listen_socket = new ServerSocket(port);
 // Задаем ненулевую паузу, чтобы accept() можно было прервать
 listen_socket.setSoTimeout(600000);
 this.port = port;
 this.service = service;
}

/**
 * Это вежливый способ сообщить Listener, что нужно прекратить
 * прием новых подключений
 ***/
public void pleaseStop() {
 this.stop = true;  // Установка флага остановки
 this.interrupt();  // Прекращение блокировки в accept().
 try { listen_socket.close(); } // Прекращение ожидания новых подключений.
 catch(IOException e) {}
}

/**
 * Класс Listener является подклассом класса Thread, его тело приведено ниже.
 * Ожидаем запросов на подключение, принимаем их и передаем Socket
 * методу сервера addConnection.
 **/
public void run() {
 while(!stop) { // цикл продолжается, пока нас не попросят остановиться.
 try {
  Socket client = listen_socket.accept();
  addConnection(client, service);
 }
```

```
 catch (InterruptedIOException e) {}
 catch (IOException e) {log(e);}
 }
}
}

/**
* Это метод, вызываемый объектами Listener, когда они принимают
* соединение с клиентом. Он либо создает объект Connection
* для этого подключения и добавляет его в список имеющихся подключений,
* либо, если лимит подключений исчерпан, закрывает подключение.
**/
protected synchronized void addConnection(Socket s, Service service) {
// Если лимит числа подключений исчерпан,
if (connections.size() >= maxConnections) {
 try {
 // сообщаем клиенту, что его запрос отклонен.
 PrintWriter out = new PrintWriter(s.getOutputStream());
 out.print("В подключении отказано; " +
   "сервер перегружен, попытайтесь подключиться позже.\n");
 out.flush();
 // И закрываем подключение клиента, которому отказано.
 s.close();
 // И, конечно, делаем об этом регистрационную запись.
 log("Подключение отклонено для " +
  s.getInetAddress().getHostAddress() +
  ":" + s.getPort() + ": исчерпан лимит числа подключений.");
 } catch (IOException e) {log(e);}
}
else { // В противном случае, если лимит не исчерпан,
       // создаем процесс Connection для обработки этого подключения.
 Connection c = new Connection(s, service);
 // Добавляем его в список текущих подключений.
 connections.add(c);
 // Регистрируем новое соединение
 log("Установлено подключение к " + s.getInetAddress().getHostAddress() +
 ":" + s.getPort() + " по порту " + s.getLocalPort() +
 " для службы " + service.getClass().getName());
 // И запускаем процесс Connection, предоставляющий услугу
 c.start();
}
}

/**
* Процесс Connection вызывает этот метод непосредственно перед выходом.
* Он удаляет заданный объект Connection из набора подключений.
**/
protected synchronized void endConnection(Connection c) {
connections.remove(c);
log("Подключение к " + c.client.getInetAddress().getHostAddress() +
 ":" + c.client.getPort() + " закрыто.");
}
```

```
/** Этот метод изменяет максимально допустимое число подключений */
public synchronized void setMaxConnections(int max) {
maxConnections = max;
}

/**
* Этот метод выводит в заданный поток информацию о статусе сервера. Он может
* применяться для отладки и ниже в этом примере используется службой Control.
**/
public synchronized void displayStatus(PrintWriter out) {
// Отображаем список всех предоставляемых служб
Iterator keys = services.keySet().iterator();
while(keys.hasNext()) {
 Integer port = (Integer) keys.next();
 Listener listener = (Listener) services.get(port);
 out.print("СЛУЖБА " + listener.service.getClass().getName()
  + " ПО ПОРТУ " + port + "\n");
}

// Отображаем текущее ограничение на число подключений
out.print("ЛИМИТ ПОДКЛЮЧЕНИЙ: " + maxConnections + "\n");

// Отображаем список всех текущих подключений
Iterator conns = connections.iterator();
while(conns.hasNext()) {
 Connection c = (Connection)conns.next();
 out.print("ПОДКЛЮЧЕНИЕ К " +
  c.client.getInetAddress().getHostAddress() +
  ":" + c.client.getPort() + " ПО ПОРТУ " +
  c.client.getLocalPort() + " ДЛЯ СЛУЖБЫ " +
  c.service.getClass().getName() + "\n");
}
}

/**
* Этот подкласс класса Thread обрабатывает индивидуальные подключения
* между клиентом и службой Service, предоставляемой настоящим сервером.
* Поскольку каждое такое подключение обладает собственным потоком
* исполнения, у каждой службы может иметься несколько подключений
* одновременно. Вне зависимости от всех других используемых потоков
* исполнения, именно это делает наш сервер многопоточным.
**/
public class Connection extends Thread {
Socket client;    // Объект Socket для общения с клиентом
Service service; // Служба, предоставляемая клиенту

/**
 * Этот конструктор просто сохраняет некоторые параметры состояния
 * и вызывает конструктор родительского класса для создания потока
 * исполнения, обрабатывающего подключение. Объекты Connection
 * создаются потоками исполнения Listener. Эти потоки являются частью группы
 * потоков сервера, поэтому процессы Connection также входят в эту группу
**/
```

```
public Connection(Socket client, Service service) {
 super("Server.Connection:" +
  client.getInetAddress().getHostAddress() +
  ":" + client.getPort());
 this.client = client;
 this.service = service;
}

/**
 * Это тело любого и каждого потока исполнения Connection. Все, что оно дела-
 * ет, - это передает потоки ввода и вывода клиента методу serve() заданного
 * объекта Service, который несет ответственность за чтение и запись
 * в эти потоки для осуществления действительного обслуживания. Вспомним,
 * что объект Service был передан методом Server.addService() объекту
 * Listener, а затем через метод addConnection() этому объекту Connection,
 * и теперь наконец используется для предоставления услуги. Обратите внимание
 * на то, что непосредственно перед выходом этот поток исполнения всегда
 * вызывает метод endConnection(), чтобы удалить себя из набора подключений.
 **/
public void run() {
 try {
 InputStream in = client.getInputStream();
 OutputStream out = client.getOutputStream();
 service.serve(in, out);
 }
 catch (IOException e) {log(e);}
 finally { endConnection(this); }
}
}

/**
* Здесь описан интерфейс Service, с которым мы так часто встречались.
* Он определяет только один метод, который вызывается для предоставления
* услуги. Методу serve() передаются поток ввода и поток вывода,
* связанные с клиентом. Он может делать с ними все, что угодно,
* только перед выходом должен закрыть их.
*
* Все соединения с этой службой по одному порту совместно используют один
* объект Service. Таким образом, любое локальное состояние индивидуального
* подключения должно храниться в локальных переменных метода serve().
* Состояние, характеризующее все подключения по данному порту, должно
* храниться в переменных экземпляра класса Service. Если одна служба Service
* запущена на нескольких портах, обычно будут иметься несколько экземпляров
* Service, по одному на каждый порт. Данные, относящиеся ко всем
* подключениям на всех портах, должны храниться в статических переменных.
*
* Обратите внимание на то, что если экземпляры этого интерфейса будут
* динамически создаваться методом main() класса Server, в их реализации
* должен быть включен конструктор без аргументов.
**/
public interface Service {
```

```
public void serve(InputStream in, OutputStream out) throws IOException;
}

/**
* Очень простая служба. Она сообщает клиенту текущее время на сервере
* и закрывает подключение.
**/
public static class Time implements Service {
public void serve(InputStream i, OutputStream o) throws IOException {
 PrintWriter out = new PrintWriter(o);
 out.print(new Date() + "\n");
 out.close();
 i.close();
}
}

/**
* Это еще один пример службы. Она считывает строки, введенные клиентом,
* и возвращает их перевернутыми. Она также выводит приветствие
* и инструкции и разрывает подключение, когда пользователь
* вводит строку, состоящую из точки «.».
**/
public static class Reverse implements Service {
public void serve(InputStream i, OutputStream o) throws IOException {
 BufferedReader in = new BufferedReader(new InputStreamReader(i));
 PrintWriter out =
 new PrintWriter(new BufferedWriter(new OutputStreamWriter(o)));
 out.print("Welcome to the line reversal server.\n");
 out.print("Enter lines. End with a '.' on a line by itself.\n");
 for(;;) {
 out.print("> ");
 out.flush();
 String line = in.readLine();
 if ((line == null) || line.equals(".")) break;
 for(int j = line.length()-1; j >= 0; j--)
  out.print(line.charAt(j));
 out.print("\n");
 }
 out.close();
 in.close();
}
}

/**
* Эта служба – просто "зеркало" HTTP, точно такое же, как класс HttpMirror,
* реализованный выше в этой главе. Она возвращает клиенту его HTTP-запросы.
**/
public static class HTTPMirror implements Service {
public void serve(InputStream i, OutputStream o) throws IOException {
 BufferedReader in = new BufferedReader(new InputStreamReader(i));
 PrintWriter out = new PrintWriter(o);
 out.print("HTTP/1.0 200 \n");
 out.print("Content-Type: text/plain\n\n");
```

```
 String line;
 while((line = in.readLine()) != null) {
 if (line.length() == 0) break;
 out.print(line + "\n");
 }
 out.close();
 in.close();
}
}

/**
* Эта служба демонстрирует, как следует поддерживать состояние на протяжении
* нескольких подключений путем сохранения его в переменных экземпляра
* и применения синхронизованного доступа к этим переменным. Эта служба
* подсчитывает число подключившихся к ней клиентов
* и сообщает каждому клиенту его номер.
**/
public static class UniqueID implements Service {
public int id=0;
public synchronized int nextId() { return id++; }
public void serve(InputStream i, OutputStream o) throws IOException {
 PrintWriter out = new PrintWriter(o);
 out.print("Ваш номер: " + nextId() + "\n");
 out.close();
 i.close();
}
}

/**
* Это нетривиальная служба. Она реализует командный протокол, дающий защищенные
* паролем средства управления операциями сервера во время его исполнения.
* См. метод main() класса Server, чтобы увидеть, как эта служба запускается.
*
* Распознаются следующие команды:
*   password: сообщает пароль; авторизация обязательна для большинства команд
*   add: динамически добавляет названную службу на заданном порте
*   remove: динамически удаляет службу, работающую на заданном порте
*   max: изменяет лимит числа подключений.
*   status: отображает действующие службы, текущие соединения
*           и лимит числа подключений
*   help: отображает страницу помощи
*   quit: отключение
*
* Эта служба выводит "подсказку" и посылает весь свой вывод в адрес клиента
* заглавными буквами. К этой службе в каждый момент времени
* может подключиться только один клиент.
**/
public static class Control implements Service {
Server server;             // Сервер, которым мы управляем
String password;           // Пароль, который мы требуем
boolean connected = false; // Подключен ли уже кто-то к этой службе?
```

```
/**
 * Создаем новую службу Control. Она будет управлять заданным объектом Server
 * и будет требовать заданный пароль для авторизации. Обратите внимание
 * на то, что у этой службы нет конструктора без аргументов. Это значит,
 * что, в отличие от вышеприведенных служб, нет возможности динамически
 * создать ее экземпляр и добавить ее к списку служб сервера.
 **/
public Control(Server server, String password) {
 this.server = server;
 this.password = password;
}

/**
 * Это метод serve(), осуществляющий обслуживание. Он читает строку,
 * отправленную клиентом, и применяет java.util.StringTokenizer, чтобы
 * разобрать ее на команды и аргументы. В зависимости от этой команды
 * он делает множество разных вещей.
 **/
public void serve(InputStream i, OutputStream o) throws IOException {
 // Настраиваем потоки
 BufferedReader in = new BufferedReader(new InputStreamReader(i));
 PrintWriter out = new PrintWriter(o);
 String line; // Для чтения строк из клиентского ввода
              // Пользователь уже сказал пароль?
 boolean authorized = false;

 // Если к этой службе уже подключен клиент, отображаем
 // сообщение для этого клиента и закрываем подключение. Мы используем
 // синхронизированный блок для предотвращения "состояния гонки".
 synchronized(this) {
 if (connected) {
  out.print("РАЗРЕШАЕТСЯ ТОЛЬКО ОДНО ПОДКЛЮЧЕНИЕ.\n");
  out.close();
  return;
 }
 else connected = true;
 }

 // Это главный цикл: в нем команда считывается,
 // анализируется и обрабатывается
 for(;;) { // бесконечный цикл
 out.print("> ");          // Отображаем "подсказку"
 out.flush();              // Выводим ее немедленно
 line = in.readLine();     // Получаем пользовательский ввод
 if (line == null) break; // Выходим из цикла при получении EOF.
 try {
  // Применяем StringTokenizer для анализа команды пользователя
  StringTokenizer t = new StringTokenizer(line);
  if (!t.hasMoreTokens()) continue; // если ввод пустой
  // Выделяем из ввода первое слово и приводим его к нижнему регистру
  String command = t.nextToken().toLowerCase();
  // Теперь сравниваем его со всеми допустимыми командами, выполняя
```

```
// для каждой команды соответствующие действия
if (command.equals("password")) { // Команда Password
String p = t.nextToken();         // Получаем следующее слово
if (p.equals(this.password)) {    // Пароль правильный?
 out.print("OK\n");               // Допустим, да
 authorized = true;               // Подтверждаем авторизацию
}
else out.print("НЕВЕРНЫЙ ПАРОЛЬ\n"); // В противном случае – нет
}
else if (command.equals("add")) { // Команда Add Service
// Проверяем, был ли указан пароль
if (!authorized) out.print("НЕОБХОДИМ ПАРОЛЬ\n");
else {
 // Получаем название службы и пытаемся динамически загрузить ее
 // и создать ее экземпляр. Исключения обрабатываются ниже
 String serviceName = t.nextToken();
 Class serviceClass = Class.forName(serviceName);
 Service service;
 try {
 service = (Service)serviceClass.newInstance();
 }
 catch (NoSuchMethodError e) {
 throw new IllegalArgumentException(
   "У службы должен быть " +
   "конструктор без аргументов");
 }
 int port = Integer.parseInt(t.nextToken());
 // Если никаких исключений не произошло, добавляем службу
 server.addService(service, port);
 out.print("СЛУЖБА ДОБАВЛЕНА\n"); // сообщаем об этом клиенту
}
}
else if (command.equals("remove")) { // Команда удаления службы
if (!authorized) out.print("НЕОБХОДИМ ПАРОЛЬ\n");
else {
 int port = Integer.parseInt(t.nextToken());
 server.removeService(port); // удаляем службу
 out.print("СЛУЖБА УДАЛЕНА\n"); // сообщаем об этом клиенту
}
}
else if (command.equals("max")) { // Устанавливаем лимит числа подключений
if (!authorized) out.print("НЕОБХОДИМ ПАРОЛЬ\n");
else {
 int max = Integer.parseInt(t.nextToken());
 server.setMaxConnections(max);
 out.print("ЛИМИТ ПОДКЛЮЧЕНИЙ ИЗМЕНЕН \n");
}
}
else if (command.equals("status")) { // Отображение состояния
if (!authorized) out.print("НЕОБХОДИМ ПАРОЛЬ\n");
else server.displayStatus(out);
}
```

```
    else if (command.equals("help")) { // Команда Help
    // Отображаем синтаксис команд. Пароль необязателен
    out.print("КОМАНДЫ:\n" +
      "\tpassword <password>\n" +
      "\tadd <service> <port>\n" +
      "\tremove <port>\n" +
      "\tmax <max-connections>\n" +
      "\tstatus\n" +
      "\thelp\n" +
      "\tquit\n");
    }
    else if (command.equals("quit")) break; // Команда Quit.
    else out.print("НЕИЗВЕСТНАЯ КОМАНДА\n"); // Ошибка
   }
   catch (Exception e) {
    // Если в процессе анализа или выполнения команды возникло исключение,
    // печатаем сообщение об ошибке, затем выводим
    // подробные сведения об исключении.
    out.print("ОШИБКА ВО ВРЕМЯ АНАЛИЗА ИЛИ ВЫПОЛНЕНИЯ КОМАНДЫ:\n" +
     e + "\n");
   }
   }
   // Окончательно, когда происходит выход из цикла обработки команд,
   // закрываем потоки (streams) и присваиваем флагу connected значение false,
   // так что теперь могут подключаться новые клиенты.
   connected = false;
   out.close();
   in.close();
  }
  }
}
```

Многопоточный прокси-сервер

Пример 5.6 показывает, как можно написать простой однопоточный прокси-сервер. Пример 5.10 использует класс `Server` и интерфейс `Server.Service`, определенные в примере 5.9, для реализации многопоточного прокси-сервера. В нем показано, как можно использовать базовый класс `Server` для реализации серверов специального назначения без применения метода `main()` класса `Server`. Тело внутреннего класса `Proxy` (релизующего интерфейс `Service`) напоминает класс `SimpleProxyServer`, но вместо одного в нем создаются два безымянных потока исполнения. Основной поток использует метод `join()` класса `Thread`, чтобы дождаться завершения обоих этих потоков.

Пример 5.10. ProxyServer.java

```
package com.davidflanagan.examples.net;
import java.io.*;
import java.net.*;
```

```
/**
 * Этот класс применяет класс Server в качестве основы многопоточного сервера,
 * на который «навешиваются» относительно простые прокси-службы. Метод main()
 * запускает сервер. Вложенный класс Proxy реализует интерфейс
 * Server.Service и предоставляет прокси-обслуживание.
 **/
public class ProxyServer {
 /**
 * Создаем объект Server и добавляем к нему объекты Proxy, осуществляющие
 * прокси-обслуживание в соответствии с аргументами, заданными
 * в командной строке.
 **/
 public static void main(String[] args) {
 try {
  // Проверяем число аргументов. Оно должно быть кратно 3 и > 0.
  if ((args.length == 0) || (args.length % 3 != 0))
  throw new IllegalArgumentException("Неправильное число аргументов");

  // Создаем объект Server
  Server s = new Server(null, 12); // Записываем поток (stream)
                                   // и лимит подключений

  // Цикл, анализирующий кортежи (tuples) аргументов (host, remoteport,
  // localport). Для каждого из них создаем объект Proxy и добавляем
  // его к списку служб сервера.
  int i = 0;
  while(i < args.length) {
  String host = args[i++];
  int remoteport = Integer.parseInt(args[i++]);
  int localport = Integer.parseInt(args[i++]);
  s.addService(new Proxy(host, remoteport), localport);
  }
 }
 catch (Exception e) { // Печатаем сообщение об ошибке,
                       // если что-то не в порядке
  System.err.println(e);
  System.err.println("Формат: java ProxyServer " +
    "<host> <remoteport> <localport> ...");
  System.exit(1);
 }
 }

 /**
 * Это класс, реализующий прокси-службу. Метод serve() будет вызываться
 * при подключении клиента. В этот момент он должен установить
 * подключение к серверу, а затем передавать байты от клиента серверу
 * и обратно. Для симметрии этот класс реализует два очень сходных потока
 * исполнения в виде безымянных классов. Один поток копирует байты
 * от клиента к серверу, а другой копирует их от сервера к клиенту.
 * Поток исполнения, вызывающий метод serve(), создает и запускает
 * эти потоки исполнения, а затем просто ожидает их завершения.
 **/
```

```
public static class Proxy implements Server.Service {
String host;
int port;

/** Запоминаем узел и порт, которые представляем */
public Proxy(String host, int port) {
 this.host = host;
 this.port = port;
}

/** Сервер вызывает этот метод при подключении клиента. */
public void serve(InputStream in, OutputStream out) {
 // Это соединения, которые мы будем использовать. Они объявлены как final,
 // поэтому их смогут использовать приведенные ниже безымянные классы.
 final InputStream from_client = in;
 final OutputStream to_client = out;
 final InputStream from_server;
 final OutputStream to_server;

 // Пытаемся установить подключение к заданному серверу и порту и получить
 // соединения для связи с ним. В случае неудачи докладываем о ней клиенту.
 final Socket server;
 try {
 server = new Socket(host, port);
 from_server = server.getInputStream();
 to_server = server.getOutputStream();
 }
 catch (Exception e) {
 PrintWriter pw = new PrintWriter(new OutputStreamWriter(out));
 pw.print("Прокси-сервер не смог подключиться к " + host +
   ":" + port + "\n");
 pw.flush();
 pw.close();
 try { in.close(); } catch (IOException ex) {}
 return;
 }

 // Создаем массив, содержащий два объекта Thread. Он объявляется как final,
 // так что им смогут пользоваться приведенные ниже безымянные классы.
 // Мы используем массив вместо двух переменных, поскольку при данной
 // структуре программы две переменные не сработали бы,
 // будучи определены как final.
 final Thread[] threads = new Thread[2];

 // Определяем и создаем поток исполнения, копирующий байты
 // от клиента к серверу
 Thread c2s = new Thread() {
  public void run() {
  // Копируем байты до тех пор, пока не получим от клиента EOF
  byte[] buffer = new byte[2048];
  int bytes_read;
  try {
   while((bytes_read=from_client.read(buffer))!=-1) {
```

```
    to_server.write(buffer, 0, bytes_read);
    to_server.flush();
    }
   }
   catch (IOException e) {}
   finally {
                       // По завершении потока исполнения
    try {
    server.close();    // закрываем соединение с сервером
    to_client.close(); // и клиентские потоки (streams)
    from_client.close();
    }
    catch (IOException e) {}
   }
   }
  };

  // Определяем и создаем поток исполнения, копирующий байты от сервера
  // к клиенту. Этот поток исполнения работает совершенно так же,
  // как приведенный выше.
  Thread s2c = new Thread() {
   public void run() {
   byte[] buffer = new byte[2048];
   int bytes_read;
   try {
    while((bytes_read=from_server.read(buffer))!=-1) {
    to_client.write(buffer, 0, bytes_read);
    to_client.flush();
    }
   }
   catch (IOException e) {}
   finally {
    try {
    server.close(); // закрываемся
    to_client.close();
    from_client.close();
    } catch (IOException e) {}
   }
   }
  };

  // Сохраняем потоки исполнения в массиве final threads[], чтобы
  // безымянные классы могли ссылаться друг на друга.
  threads[0] = c2s; threads[1] = s2c;

  // запускаем потоки исполнения
  c2s.start(); s2c.start();

  // Ожидаем их завершения
  try { c2s.join(); s2c.join(); } catch (InterruptedException e) {}
 }
 }
}
```

Отправка дейтаграмм

Теперь, когда мы тщательно изучили возможности сетевых операций с соединениями и потоками, давайте посмотрим, как могут осуществляться сетевые операции низкого уровня с применением дейтаграмм и пакетов. Примеры 5.11 и 5.12 показывают, как можно организовать простую передачу данных по сети с применением дейтаграмм. Сообщение посредством дейтаграмм иногда обозначают UDP (Unreliable Datagram Protocol, ненадежный протокол дейтаграмм). Отправляются дейтаграммы быстро, но проигрыш при их использовании состоит в том, что нет гарантии, что они попадут по назначению. Кроме того, несколько дейтаграмм необязательно проходят к пункту назначения по одному маршруту и необязательно будут поступать в пункт назначения в том порядке, в котором они были отправлены. Дейтаграммы полезны в тех случаях, когда нужно снизить издержки при обмене данными, не имеющими критически важного значения, и когда потоковая модель коммуникации не является обязательной. Например, с применением дейтаграмм можно реализовать многопользовательский диалоговый (chat) сервер для локальной сети.

Для отправки и получения дейтаграмм используются классы `DatagramPacket` и `DatagramSocket`. Эти объекты создаются и инициализируются по-разному в зависимости от того, отправляете вы или принимаете дейтаграммы. Пример 5.11 показывает, как отправлять дейтаграммы. Пример 5.12 показывает, как принимать дейтаграммы и как узнать, кем они были отправлены.

Чтобы отправить дейтаграмму, сначала создается объект `DatagramPacket`, содержащий передаваемые данные, их длину, узел и порт этого узла, на которые посылается дейтаграмма. Затем для отправки пакета применяется метод `send()` объекта `DatagramSocket`. Объект `DatagramSocket` является универсальным, он создается без аргументов. Он может использоваться многократно для отправки любого пакета по любому адресу и порту.

Пример 5.11. UDPSend.java

```
package com.davidflanagan.examples.net;
import java.io.*;
import java.net.*;

/**
 * Этот класс отправляет заданный текст или файл в виде дейтаграммы
 * на заданный порт заданного узла.
 **/
public class UDPSend {
 public static final String usage =
 "Формат: java UDPSend <hostname> <port> <msg>...\n" +
 " или: java UDPSend <hostname> <port> -f <file>";

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

```
 try {
  // Проверяем число аргументов
  if (args.length < 3)
  throw new IllegalArgumentException("Неправильное число аргументов");

  // Разбираем аргументы
  String host = args[0];
  int port = Integer.parseInt(args[1]);

  // Определяем пересылаемое сообщение.
  // Если третий аргумент -f, отправляем содержимое файла,
  // заданного четвертым аргументом. В противном случае объединяем
  // третий и все последующие аргументы и отправляем их.
  byte[] message;
  if (args[2].equals("-f")) {
  File f = new File(args[3]);
  int len = (int)f.length(); // Определяем размер файла
  message = new byte[len];   // Создаем буфер достаточного размера
  FileInputStream in = new FileInputStream(f);
  int bytes_read = 0, n;
  do {    // Цикл продолжается, пока не все прочитано
   n = in.read(message, bytes_read, len-bytes_read);
   bytes_read += n;
  } while((bytes_read < len) && (n != -1));
  }
  else { // В противном случае просто суммируем все остальные аргументы.
  String msg = args[2];
  for (int i = 3; i < args.length; i++) msg += " " + args[i];
  message = msg.getBytes();
  }

  // Получаем адрес заданного узла в Интернете
  InetAddress address = InetAddress.getByName(host);

  // Инициализируем пакет дейтаграммы данными и адресом
  DatagramPacket packet = new DatagramPacket(message, message.length,
       address, port);

  // Создаем "дейтаграммное" соединение, отправляем по нему пакет, закрываем его
  DatagramSocket dsocket = new DatagramSocket();
  dsocket.send(packet);
  dsocket.close();
 }
 catch (Exception e) {
  System.err.println(e);
  System.err.println(usage);
 }
 }
}
```

Прием дейтаграмм

Пример 5.12 – это программа, которая сидит и ждет прибытия дейтаграмм. Приняв таковую, она печатает содержимое дейтаграммы и имя узла, с которого дейтаграмма была отправлена.

Чтобы принять дейтаграмму, необходимо сначала создать объект `DatagramSocket`, слушающий определенный порт локального узла. Это соединение способно принимать пакеты, направленные в адрес этого конкретного порта. Затем необходимо создать `DatagramPacket` с буфером байтов, в котором будет сохраняться содержимое дейтаграммы. Наконец, следует вызвать метод `DatagramSocket.receive()`, который будет ожидать прибытия дейтаграммы на заданный порт. Когда это произойдет, содержащиеся в ней данные переместятся в заданный буфер, и метод `receive()` завершится. Если дейтаграмма длиннее заданного буфера, лишние байты пропадут. По прибытии дейтаграммы `receive()` также сохраняет узел и порт, с которых дейтаграмма была отправлена, в объекте `DatagramPacket`.

Пример 5.12. UDPReceive.java

```
package com.davidflanagan.examples.net;
import java.io.*;
import java.net.*;

/**
 * Эта программа ожидает прибытия дейтаграмм, отправленных на заданный порт.
 * Приняв дейтаграмму, она отображает узел отправителя и выводит содержимое
 * дейтаграммы в виде строки. Затем она возвращается к началу
 * цикла и снова ждет.
 **/
public class UDPReceive {
 public static final String usage = "Формат: java UDPReceive <port>";
 public static void main(String args[]) {
 try {
  if (args.length != 1)
  throw new IllegalArgumentException("Неправильное число аргументов");

  // Получаем порт из командной строки
  int port = Integer.parseInt(args[0]);

  // Создаем соединение, слушающее этот порт.
  DatagramSocket dsocket = new DatagramSocket(port);

  // Создаем буфер для считывания дейтаграммы. Если кто-то отправит нам пакет
  // большего размера, чем может поместиться в буфере, избыток просто пропадет!
  byte[] buffer = new byte[2048];

  // Создаем пакет для приема данных в буфер
  DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

  // Запускаем бесконечный цикл, ожидающий получения пакетов и печатающий их.
  for(;;) {
```

```
        // Ждем прибытия дейтаграммы
        dsocket.receive(packet);

        // Преобразуем ее содержимое в объект String и отображаем его
        String msg = new String(buffer, 0, packet.getLength());
        System.out.println(packet.getAddress().getHostName() +
            ": " + msg);

        // Перед следующим использованием пакета packet восстанавливаем его длину.
        // До появления Java 1.1 нам пришлось бы каждый раз создавать новый пакет.
        packet.setLength(buffer.length);
        }
      }
      catch (Exception e) {
        System.err.println(e);
        System.err.println(usage);
      }
      }
    }
```

Упражнения

5-1. Применяя приемы работы с классом `URLConnection`, показанные в примере 5.2, напишите программу, печатающую дату модификации заданного URL.

5-2. Преобразуйте программу `HttpClient` из примера 5.4 так, чтобы она применяла более свежую версию протокола HTTP. Для этого вам придется отправить на веб-сервер несколько более сложный запрос `GET`. Примените программу `HttpMirror` из примера 5.5, чтобы узнать, какую форму принимает этот запрос. HTTP Versions 1.0 и более поздних версий добавляет к запросу `GET` номер версии, а за строкой `GET` помещает множество заголовочных строк и пустую строку как признак конца запроса.

В этом упражнении вам понадобится добавить только одну строку заголовка – строку User-Agent, идентифицирующую используемый вами веб-клиент. Поскольку вы пишете ваш собственный веб-клиент, можно дать ему любое понравившееся имя! Не забудьте поместить вслед за запросом `GET` и заголовком User-Agent пустую строку, иначе веб-сервер будет продолжать ждать следующие заголовки и никогда не ответит на ваш запрос. Когда ваша программа заработает, вы должны заметить, что веб-серверы отвечают на ее запросы иначе, чем они отвечали на запросы первоначальной программы `HttpClient`. Когда клиент запрашивает данные с применением HTTP 1.0 или 1.1, перед отправкой запрашиваемого файла сервер посылает номер версии, код состояния и множество заголовочных строк ответа.

5-3. Напишите простой сервер, сообщающий время дня (в текстовой форме) всякому подключившемуся клиенту. Используйте пример 5.5, `HttpMirror`, в качестве основы вашего сервера. Этот сервер должен просто вывести текущее время и закрыть подключение, не читая ничего, что поступило от клиента. Вам придется выбрать номер порта, по которому ваша служба будет ожидать подключений. Используйте программу `GenericClient` из примера 5.8 для подключения к этому порту и тестирования программы. В качестве альтернативы можно использовать программу `HttpClient` из примера 5.4 для тестирования вашей программы. Для этой цели включите подходящий номер порта в URL. `HttpClient` отправит серверу нестандартный запрос `GET`, но все же отобразит ответ сервера.

5-4. В обсуждении программы `GenericClient` из примера 5.8 содержался пример использования этой программы для коммуникации с сервером POP (Post Office Protocol) для получения электронной почты. Протокол POP очень прост; поэкспериментировав с `GenericClient`, вы быстро поймете, как он работает. (Постарайтесь не удалить при этом важную электронную почту!)

В этом упражнении вам предлагается написать программу-клиент `Checkmail`, использующую протокол POP для проверки почты пользователя. Она должна выводить число сообщений, ожидающих прочтения, и отображать строку From каждого послания. Этот клиент *не* должен использовать команду POP `dele` для удаления с сервера почтовых сообщений; он должен только выводить сводку по сообщениям, ожидающим получения. Чтобы получить возможность прочитать сообщения, эта программа должна знать имя узла POP-сервера и должна послать на сервер имя пользователя и пароль. Ваша программа должна получать имя узла, имя пользователя и пароль из командной строки или запрашивать их у пользователя, но в идеале она должна получать эту информацию из файла конфигурации. Рассмотрите применение объекта `java.util.Properties` для реализации такого конфигурационного файла.

5-5. Напишите простой веб-сервер, отвечающий на запросы `GET`, подобные тем, что генерируются программой, которую вы написали в упражнении 5.2. Возможно, вы захотите использовать программу `HttpMirror` из примера 5.5 как основу для вашего сервера. Иначе вы можете реализовать ваш сервер как подкласс `Service` для применения с программой `Server`, разработанной в примере 5.9.

Ваш сервер должен пользоваться протоколом HTTP 1.0 или более поздних версий. Это значит, что сервер ожидает заголовочных строк, следующих за запросом `GET` и завершаемых пустой строкой. Также это значит, что ваши ответы на запросы `GET` должны начинаться с номера версии и кода состояния. За этой строкой состояния следуют заголовочные строки, завершающиеся пустой

строкой. Содержание ответа следует за пустой строкой. Примените написанную вами в упражнении 5.2 клиентскую программу для экспериментов с существующими веб-серверами, чтобы посмотреть, как они отвечают на различные запросы `GET`.

Веб-сервер, который вы пишете, должен запускаться с каталогом, заданным в командной строке, и должен искать файлы относительно этого каталога. Когда клиент запрашивает файл из этого каталога или его подкаталогов, сервер должен возвратить содержимое файла, но сначала он должен вывести строки заголовков Content-Type, Content-Length и Last-Modified. В этом упражнении можно допустить, что файлы с расширениями *.html* или *.htm* имеют тип содержимого `text/html`, а все другие файлы – `text/plain`. Если клиент запрашивает несуществующий файл, ваш сервер должен вывести соотвествующие код ошибки и сообщение об ошибке. И снова воспользуйтесь клиентом, разработанным в упражнении 5-2, чтобы узнать, как существующие веб-серверы отвечают на запрос несуществующего файла.

5-6. Преобразуйте программы `UDPSend` и `UDPReceive` из примеров 5.11 и 5.12 так, чтобы `UDPReceive` отправляла уведомление о приеме дейтаграммы, а `UDPSend` не прекращала работы до получения уведомления. Уведомление само должно быть дейтаграммой и может содержать любые данные, какие вы захотите. (Вы можете, например, воспользоваться классами, предназначенными для работы с контрольной суммой, из пакета `java.util.zip` для вычисления контрольной суммы принятых данных и вернуть ее в составе пакета уведомления.) Используйте метод `setSoTimeout()` класса `DatagramSocket`, чтобы программа `UDPSend` ожидала уведомления не более нескольких секунд. Если в течение этого времени уведомление не поступает, `UDPSend` должна предположить, что первоначальный пакет потерян и не был принят. Ваша модифицированная программа `UDPSend` должна раз или два повторить попытку и, если уведомление так и не получено, завершить работу с выдачей сообщения об ошибке.

Глава 6

Защита и криптография

Средства защиты – одна из главных особенностей, сделавших язык Java столь успешным. Система безопасности Java включает в себя механизмы управления доступом, позволяющие ненадежным программам, таким как апплеты, исполняться в безопасном режиме, без риска что-то повредить, украсть секреты фирмы или нанести какой-либо другой ущерб. Применяемые Java механизмы управления доступом претерпели значительные изменения на пути от Java 1.0 к Java 1.2; в этой главе мы обсудим механизмы безопасности Java 1.2.

Управление доступом составляет, однако, только половину системы безопасности Java. Вторая половина – это аутентификация (проверка подлинности). Пакет `java.security` и его подпакеты позволяют создавать и проверять зашифрованные контрольные суммы и цифровые подписи, дающие возможность удостовериться, что файл класса Java (или любой другой файл) является подлинным, то есть действительно имеет то происхождение, которое он себе приписывает. Вместе с прогрессом Java API аутентификации также видоизменялся, и я здесь охватываю API, принадлежащий Java 1.2.

Управление доступом и проверка подлинности в системе безопасности Java тесно связаны. Управление доступом должно предоставлять привилегии только заслуживающему доверия коду. Но какой код заслуживает доверия? Если вы знаете, каким людям или организациям можно доверять (что, в конце концов, является социальной, а не технологической проблемой), вы можете применить механизмы проверки подлинности, такие как цифровые подписи, чтобы оказывать доверие файлам классов Java, поступившим от этих людей и организаций.

Криптография представляет собой важную составляющую системы безопасности Java. Вследствие строгих правил экспорта, принятых в США, шифровальные технологии не входят в стандартную поставку

Java. Однако для поддержки шифрования и дешифрования доступен, в виде расширения, пакет Java Cryptography Extension™ (JCE).

В этой главе содержатся примеры, показывающие, как для управления доступом, проверки подлинности и шифрования могут применяться Java API и JCE.

Исполнение ненадежного кода

Вспомним пример `Server` из главы 5 «Сетевые операции». Этот класс универсального сервера динамически загружал и запускал реализации интерфейса `Service`. Представьте, что вы администратор сервера, отвечающий за программу `Server`, и что вы не доверяете программистам, разрабатывающим службы сервера; вы боитесь, что они случайно (или злонамеренно) включили в свои классы `Service` вредоносный код. Java дает удобную возможность запуска этих подозрительных классов под наблюдением механизмов управления доступом, способных помешать этим классам совершить что-либо недозволенное.

Управление доступом в Java осуществляют классы `SecurityManager` (диспетчер безопасности) и `AccessController`. После того как диспетчер безопасности зарегистрирован, Java сверяется с ним всякий раз, когда его просят совершить какую-нибудь операцию (такую как чтение или запись файла, или сетевое подключение), на доступ к которой могут быть наложены ограничения. В Java 1.2 и более поздних версий класс `SecurityManager` использует класс `AccessController` для осуществления таких проверок прав доступа, а `AccessController`, в свою очередь, сверяется с файлом `Policy`, в котором в точности указано, какие объекты `Permission` (права доступа) какому коду даны.

В Java 1.2 запустить код так, чтобы он исполнялся под постоянным контролем диспетчера безопасности, очень просто. Следует просто запустить интерпретатор Java с опцией `-D` для установки свойства `java.security.manager`. Например, запустить класс `Server` под надзором диспетчера безопасности можно командой:

```
% java -Djava.security.manager com.davidflanagan.examples.net.Server \
-control password 4000
```

Если вы это сделаете, и класс `Server`, и загружаемый им класс управляющей службы будут подвергаться проверкам управления доступом со стороны диспетчера безопасности, который будет применять принятые в системе по умолчанию политики безопасности (security policy).

Если вы попытаетесь запустить `Server` с применением принятых по умолчанию политик безопасности, поставляемых корпорацией Sun, сервер потерпит неудачу при первой попытке подключения клиента, и вы увидите следующее сообщение:

```
java.security.AccessControlException:
     access denied (java.net.SocketPermission 127.0.0.1:1170 accept,resolve)
```

Это сообщение говорит вам, что диспетчер безопасности не разрешил классу `Server` принять сетевое подключение клиента. Причина здесь в том, что принимаемые по умолчанию политики безопасности являются слишком строгими для нашего сервера. К счастью, существует простой способ разрешить серверу принимать подключения. Следует создать файл *Server.policy* следующего содержания (нужно только заменить название каталога именем того каталога, в котором установлены ваши классы):

```
// Эти строки дают права доступа любому коду, загруженному из указанного каталога
// Исправьте его название в соответствии с установками в вашей системе.
// В системах Windows замените прямой слэш на двойной обратный слэш (\\).
grant codeBase "file:/home/david/Books/JavaExamples2/Examples" {
          // Разрешаем серверу ожидать и принимать сетевые подключения
          // с любого узла на порт > 1024
         permission java.net.SocketPermission "*:1024-", "listen,accept"; };
```

После создания файла *Server.policy* снова запустите класс `Server`, но теперь с еще одной опцией `-D`, указывающей, что интерпретатор должен использовать этот файл политик (policy file):

```
% java -Djava.security.manager -Djava.security.policy=Server.policy \
com.davidflanagan.examples.net.Server -control password 4000
```

При задании такой командной строки интерпретатор Java берет принимаемые по умолчанию политики безопасности и добавляет к ним политики, заданные в командной строке. Обратите внимание на то, что при использовании в командной строке `==` вместо `=` интерпретатор проигнорирует принимаемые по умолчанию политики и будет руководствоваться только политиками, заданными вами. Наш файл *Server.policy* сработает в обоих случаях.

Мораль тут такова: если вы написали Java-приложение и хотите, чтобы его запускали люди, не имеющие оснований вам доверять, вы должны представить себе, какого рода небезопасные действия оно предпринимает, и создать для него файл политик. Тогда ваши пользователи смогут изучить файл политик, чтобы увидеть, каких прав доступа требует приложение. Если они захотят дать вашему коду эти права доступа, они смогут запустить вашу программу с показанной выше опцией `-D`, зная наверняка, что ваш код не предпримет никаких опасных действий, кроме явно дозволяемых ему файлом политик.

Чтобы вполне разобраться в механизмах управления доступом в Java, вам понадобится прочесть о классе `java.security.Permission` и о множестве его подклассов. Вам придется также почитать о классе `java.security.Policy`. Чтобы освоить создание собственных файлов политик, вам следует познакомиться с программой *policytool*, поставляемой корпорацией Sun вместе с Java SDK. Загляните в «Java in a Nutshell»[1].

[1] Дэвид Флэнаган «Java. Справочник», 4-е издание, Символ-Плюс, 2003.

Если вы захотите редактировать файлы политик вручную (часто это удобнее всего), подробности о формате файла вы сможете найти в поставляемой вместе с SDK документации по системе безопасности.

Загрузка ненадежного кода

Продолжим рассмотрение примера `Server`. Предположим теперь, что вы хотите так преобразовать ваш сервер, чтобы он мог загружать классы `Service` с произвольного URL в сети. Предположим теперь, что вы хотите дать классам `Service` возможность читать и записывать файлы из временного каталога локальной системы. Вы можете добиться этого, написав простой класс, использующий `URLClassLoader` для загрузки классов служб и передачи их экземпляру класса `Server`. Однако чтобы заставить все это работать, вам придется также создать подходящий файл политик.

Пример 6.1 демонстрирует такой класс `SafeServer`. Как и оригинальному классу `Server`, ему в командной строке передается список классов `Service` и номеров портов. Но в качестве первого аргумента в командной строке он будет ожидать значение URL, с которого должны быть загружены классы служб.

Пример 6.1. SafeServer.java

```
package com.davidflanagan.examples.security;
import com.davidflanagan.examples.net.Server;
import java.io.*;
import java.net.*;
import java.security.*;

/**
 * Этот класс является программой, использующей созданный в главе 5
 * класс Server. Server для предоставления услуг загружает произвольные
 * классы служб Service. Этот класс является альтернативной программой,
 * запускающей Server подобным образом. Разница в том, что эта программа
 * использует SecurityManager и ClassLoader, чтобы помешать классам Service
 * совершать разрушительные или другие вредоносные действия в локальной
 * системе. Это позволяет безопасно использовать классы Service,
 * поступившие из ненадежных источников.
 **/
public class SafeServer {
  public static void main(String[] args) {
    try {
      // Устанавливаем диспетчер безопасности, если пользователь еще
      // не установил его,
      // при помощи аргумента -Djava.security.manager
      if (System.getSecurityManager() == null) {
        System.out.println("Устанавливается диспетчер безопасности");
        System.setSecurityManager(new SecurityManager());
      }
```

```
        // Создаем объект Server
        Server server = new Server(null, 5);

        // Создаем ClassLoader, который будем использовать для загрузки
        // классов Service. Классы следует сохранять в JAR-файле или в каталоге,
        // заданном первым аргументом в командной строке в виде URL
        URL serviceURL = new URL(args[0]);
        ClassLoader loader =
          new java.net.URLClassLoader(new URL[] {serviceURL});

        // Разбираем список аргументов, которые должны быть парами
        // «имя службы/порт». Для каждой пары при помощи загрузчика классов
        // загружаем названную службу, затем при помощи newInstance() создаем
        // ее экземпляр и приказываем серверу запустить ее.
        int i = 1;
        while(i < args.length) {
          // Динамически загружаем класс Service, используя загрузчик классов
          Class serviceClass = loader.loadClass(args[i++]);
          // Динамически создаем экземпляр класса.
          Server.Service service =
            (Server.Service)serviceClass.newInstance();
          int port = Integer.parseInt(args[i++]); // Выделяем номер порта
          server.addService(service, port);       // Запускаем службу
        }
      }
      catch (Exception e) { // Если что-то оказалось не в порядке,
                            // выводим сообщение
        System.err.println(e);
        System.err.println("Формат: java " + SafeServer.class.getName() +
                   " <url> <servicename> <port>\n" +
                   "\t[<servicename> <port> ... ]");
        System.exit(1);
      }
    }
  }
```

Политики безопасности для SafeServer

Класс `SafeServer` **создает и устанавливает** `SecurityManager`**, даже если пользователь не сделал этого при помощи аргумента** `-Djava.security.manager`**. Это означает, что данная программа не будет работать при отсутствии политик безопасности, дающих ей необходимые права доступа. Пример 6.2 предъявляет файл политик, который можно использовать, чтобы заставить программу работать.**

Необходимо сделать еще пару замечаний по поводу файла *SafeServer.policy*. Во-первых, файл политик читает системные свойства `service.dir` **и** `service.tmp`**. Это не стандартные системные свойства; это свойства, которые необходимо задать интерпретатору Java при запуске программы** `SafeServer`**. Свойство** `service.dir` **задает каталог, из которого должны загружаться классы служб. Будем предполагать здесь, что они за-**

гружаются с локального *file:* URL, а не с *http:* или другого сетевого URL. Свойство `service.tmp` задает каталог, в котором классы служб могут читать и записывать временные файлы. Пример *SafeServer.policy* демонстрирует синтаксис замены имени системного свойства его значением. Таким образом, можно сделать файл политик до известной степени не зависимым от места установки приложения.

Пример 6.2. SafeServer.policy

```
// Этот файл дает классу SafeServer права доступа, нужные ему, чтобы
// загружать классы Service при помощи URLClassLoader, а также разрешает
// классам Service читать и записывать файлы в заданном системным
// свойством service.tmp каталоге (и его подкаталогах). Обратите внимание
// на то, что вам придется исправить URL, задающий расположение SafeServer,
// и что в системах Windows необходимо заменить «/» на «\\»

// Даем права доступа классу SafeServer. Исправьте каталог для своей системы.
grant codeBase "file:/home/david/Books/JavaExamples2/Examples" {
   // Разрешаем серверу ожидать и принимать подключения
   // с любого узла на любом порте > 1024
   permission java.net.SocketPermission "*:1024-", "listen,accept";

   // Разрешаем серверу создать загрузчик классов для загрузки классов служб
   permission java.lang.RuntimePermission "createClassLoader";

   // Разрешаем серверу читать каталог, содержащий классы служб.
   // Если бы использовался сетевой URL, а не локальный (file:) URL,
   // нужно было бы задать SocketPermission вместо FilePermission
   permission java.io.FilePermission "${service.dir}/-", "read";

   // Сервер не может давать права доступа классам Service, пока у него
   // самого нет этих прав доступа. Поэтому дадим серверу
   // права доступа классов Service.
   permission java.util.PropertyPermission "service.tmp", "read";
   permission java.io.FilePermission "${service.tmp}/-", "read,write";
};

// Даем права доступа классам, загруженным из каталога, заданного
// системным свойством service.dir. Если бы использовался сетевой URL,
// а не локальный (file:) URL, эта строка выглядела бы иначе.
grant codeBase "file:${service.dir}" {
   // Службам разрешается считывать системное свойство «service.tmp»
   permission java.util.PropertyPermission "service.tmp", "read";
   // Также они могут читать и записывать файлы в каталоге, заданном
   // этим системным свойством
   permission java.io.FilePermission "${service.tmp}/-", "read,write";
};
```

Тестирование SafeServer

Чтобы показать, что `SafeServer` выполняет свои службы безопасно, вам понадобится демонстрационная служба. Пример 6.3 предъявляет одну такую службу, предпринимающую попытки совершить различные ограниченные системой безопасности действия и сообщающую о резуль-

татах. Обратите внимание, что поскольку вы собираетесь загружать этот класс при помощи специально созданного загрузчика классов, а не заданием пути к классам, я могу не беспокоиться о написании оператора package.

Пример 6.3. SecureService.java

```
import com.davidflanagan.examples.net.*; // Обратите внимание на отсутствие
                                         // здесь оператора package.
import java.io.*;

/**
 * Это демонстрационная служба. Она пытается совершать действия,
 * которые могут быть дозволены ей политикой безопасности, а могут
 * и не быть дозволены, и сообщает клиенту о результате этих попыток.
 **/
public class SecureService implements Server.Service {
  public void serve(InputStream i, OutputStream o) throws IOException {
    PrintWriter out = new PrintWriter(o);

    // Пытаемся установить наш собственный диспетчер безопасности. Если бы
    // нам это удалось, мы бы избавились от всякого управления доступом.
    out.println("Пытаюсь создать и установить диспетчер безопасности...");
    try {
      System.setSecurityManager(new SecurityManager());
      out.println("Получилось!");
    }
    catch (Exception e) { out.println("Failed: " + e); }

    // Пытаемся заставить Server и Java VM прекратить работу. Это запрещенное
    // для службы действие, и оно не должно успешно выполниться!
    out.println();
    out.println("Пытаюсь выйти...");
    try { System.exit(-1); }
    catch (Exception e) { out.println("Failed: " + e); }

    // Принимаемая по умолчанию системная политика позволяет
    // читать это свойство
    out.println();
    out.println("Пытаюсь узнать версию java...");
    try { out.println(System.getProperty("java.version")); }
    catch (Exception e) { out.println("Failed: " + e); }

    // Принимаемая по умолчанию системная политика не позволяет
    // читать это свойство
    out.println();
    out.println("Пытаюсь узнать домашний каталог...");
    try { out.println(System.getProperty("user.home")); }
    catch (Exception e) { out.println("Failed: " + e); }

    // Наш файл политик явным образом разрешает читать это свойство
    out.println();
    out.println("Пытаюсь прочитать свойство service.tmp...");
    try {
```

```
        String tmpdir = System.getProperty("service.tmp");
        out.println(tmpdir);
        File dir = new File(tmpdir);
        File f = new File(dir, "testfile");

        // Проверяем, действительно ли мы имеем право писать в каталог tmpdir
        out.println();
        out.println("Пытаюсь записать в " + tmpdir + "...");
        try {
          new FileOutputStream(f);
          out.println("Для записи открыт файл: " + f);
        }
        catch (Exception e) { out.println("Failed: " + e); }

        // Проверяем, действительно ли мы имеем право читать файлы
        // из каталога tmpdir
        out.println();
        out.println("Пытаюсь прочитать из " + tmpdir + "...");
        try {
          FileReader in = new FileReader(f);
          out.println("Для чтения открыт файл: " + f);
        }
        catch (Exception e) { out.println("Failed: " + e); }
      }
      catch (Exception e) { out.println("Failed: " + e); }

      // Закрываем соединения Service
      out.close();
      i.close();
    }
  }
```

Для тестирования `SafeServer` при помощи класса `SecureService` нужно решить, какие каталоги будут использоваться для хранения классов служб и для временных файлов. Ниже в тексте я использовал в качестве этих имен каталогов */tmp/services* и */tmp/scratch* соответственно.

Прежде всего, откомпилируем `SecureService` с опцией `-d`, чтобы сообщить *javac*, куда поместить получившийся файл класса:

```
% javac -d /tmp/services SecureService.java
```

Важно убедиться, что в текущем каталоге или где-нибудь еще, куда Java может добраться по локальному пути к классам, нет копии файла *SecureService.class*. Если `URLClassLoader` может обнаружить класс локально, он не станет утруждать себя загрузкой класса с заданного вами URL.

Теперь, чтобы запустить класс `SafeServer`, задайте имя класса `SecureService`, URL, с которого его следует загрузить, порт, по которому он будет ожидать подключений, и четыре разных системных свойства при помощи опций `-D`:

```
% java -Djava.security.manager -Djava.security.policy=SafeServer.policy \
   -Dservice.dir=/tmp/services -Dservice.tmp=/tmp/scratch \
   com.davidflanagan.examples.security.SafeServer file:/tmp/services/ \
   SecureService 4000
```

Командная строка получилась сложной, но она приводит к желаемому результату. Подключившись к порту 4000, вы получите от службы следующий вывод:

```
% java com.davidflanagan.examples.net.GenericClient localhost 4000
Connected to localhost/127.0.0.1:4000
Пытаюсь создать и установить диспетчер безопасности...
Failed: java.security.AccessControlException: access denied
(java.lang.RuntimePermission createSecurityManager)

Пытаюсь выйти...
Failed: java.security.AccessControlException: access denied
 (java.lang.RuntimePermission exitVM)

Пытаюсь узнать версию java...
1.3.0

Пытаюсь узнать домашний каталог...
Failed: java.security.AccessControlException: access denied
 (java.util.PropertyPermission user.home read)

Пытаюсь прочитать свойство service.tmp...
/tmp/scratch

Пытаюсь записать в /tmp/scratch...
Для записи открыт файл: /tmp/scratch/testfile

Пытаюсь прочитать из /tmp/scratch...
Для чтения открыт файл: /tmp/scratch/testfile
Connection closed by server.
```

Дайджесты сообщений и цифровые подписи

Раздел проверки подлинности в Java Security API включает поддержку дайджестов сообщения (message digest, известных также как зашифрованные контрольные суммы, cryptographic checksum), цифровых подписей (digital signature) и простых задач администрирования шифровальных ключей через абстракцию хранилища ключей («keystore»). Пример 6.4 представляет программу `Manifest`, демонстрирующую применение дайджестов сообщения, цифровых подписей и хранилищ ключей. Программа `Manifest` обеспечивает следующие функциональные возможности:

- Если передать ей в командной строке список файлов, программа прочитает каждый файл, вычислит для содержимого файла дайджест сообщения и сделает запись в файле манифеста (по умолчанию

называющемся *MANIFEST*), задающую имя каждого файла и его дайджест.

- Если вы примените необязательный флаг `-s` для задания генератора подписей (sigher) и флаг `-p` для задания пароля, программа подпишет файл манифеста и включит в него цифровую подпись.

- При вызове программы с опцией `-v` она проверит существующий файл манифеста. Сначала она проверит цифровую подпись, если таковая имеется. Если подпись действительна, она прочитает все файлы, упомянутые в манифесте, и проверит, совпадают ли их дайджесты с заданными в манифесте.

Применение программы `Manifest` для создания подписанного файла манифеста и последующей его проверки преследует две цели. Во-первых, дайджесты сообщений гарантируют, что указанные файлы не были случайно или злонамеренно изменены или испорчены после вычисления дайджеста. И во-вторых, цифровая подпись гарантирует, что сам файл манифеста не изменялся с момента подписания. (Присоединение цифровой подписи к файлу подобно подписанию юридического документа. Подписывая файл манифеста, вы фактически утверждаете, что содержимое манифеста истинно и действительно и что вы готовы ручаться за его достоверность.)

Цифровые подписи при шифровании используют технологию открытых ключей. *Закрытый ключ* создает цифровую подпись, а *открытый ключ* проверяет подпись. Классы из пакета `java.security` рассчитывают, что они смогут найти эти ключи в некоем хранилище ключей. В этой базе данных хранятся всевозможные ключи, принадлежащие лицам, организациям, компьютерам или программам.

Чтобы заставить этот пример работать, вам понадобится сгенерировать пару ключей (открытый и закрытый) для себя (или для некоторого субъекта) и поместить эти ключи в хранилище ключей. В Java SDK входит программа *keytool*, которую можно использовать для генерирования ключей и осуществления других операций с хранилищем ключей. Примените *keytool* описанным ниже способом для генерирования пары ключей для себя. Обратите внимание на то, что программа запрашивает нужную ей информацию, включая пароли. В книге «Java in a Nutshell» есть документация на программу *keytool*.

```
% keytool -genkey -alias david
Enter keystore password: secret
What is your first and last name?
  [Unknown]: David Flanagan
What is the name of your organizational unit?
  [Unknown]:
What is the name of your organization?
  [Unknown]: davidflanagan.com
What is the name of your City or Locality?
  [Unknown]: Bellingham
```

```
What is the name of your State or Province?
  [Unknown]: WA
What is the two-letter country code for this unit?
  [Unknown]: US
Is &lt;CN=David Flanagan, OU=Unknown, O=davidflanagan.com, L=Bellingham, ST=WA,
C=US&gt; correct?
  [no]: yes

Enter key password for &lt;david&gt;
  (RETURN if same as keystore password): moresecret
```

Пример 6.4 использует классы `MessageDigest` **и** `DigestInputStream` **для вычисления и проверки дайджестов сообщений. Он применяет класс** `Signature` **с** `PrivateKey` **для вычисления цифровых подписей, а также класс** `Signature` **с** `PublicKey` **для проверки цифровых подписей. Объекты** `PrivateKey` **и** `PublicKey` **получаются из объекта** `KeyStore`**. Сам файл манифеста создается и читается объектом** `java.util.Properties`**, идеально подходящим для этой цели. Дайджесты сообщений и цифровые подписи сохраняются в файле манифеста в простой шестнадцатеричной кодировке, реализованной вспомогательными методами, описанными в конце примера. (Это один их недостатков пакета** `java.security`**; он не предоставляет удобного способа преобразования массива байтов в переносимое текстовое представление.)**

Пример 6.4. Manifest.java

```
package com.davidflanagan.examples.security;
import java.security.*;
import java.io.*;
import java.util.*;

/**
 * Эта программа создает файл манифеста для заданных файлов или проверяет
 * существующий файл манифеста. По умолчанию файл манифеста называется
 * MANIFEST, но с помощью опции -m его можно назвать иначе. Опция -v
 * указывает, что файл следует проверить. Проверка также является
 * действием, совершаемым, когда никакие файлы не заданы.
 **/
public class Manifest {
 public static void main(String[] args) throws Exception {
  // Устанавливаем принимаемые по умолчанию значения аргументов
  // командной строки
  boolean verify = false;     // Проверяем манифест или создаем?
  String manifestfile = "MANIFEST"; // Имя файла манифеста
  String digestAlgorithm = "MD5";  // Алгоритм создания дайджестов сообщений
  String signername = null;   // Генератор подписей. По умолчанию отсутствует
  String signatureAlgorithm = "DSA"; // Алгоритм для цифровой подписи
  String password = null;     // Закрытые ключи защищены
  File keystoreFile = null;   // Где ключи хранятся
  String keystoreType = null;   // Какого вида это хранилище
  String keystorePassword = null;  // Каков доступ к хранилищу
  List filelist = new ArrayList(); // Файлы, для которых создается дайджест
```

```
// Анализируем аргументы, заданные в командной строке,
// переопределяя значения вышеприведенных переменных
for(int i = 0; i < args.length; i++) {
 if (args[i].equals("-v")) verify = true;
 else if (args[i].equals("-m")) manifestfile = args[++i];
 else if (args[i].equals("-da") && !verify)
  digestAlgorithm = args[++i];
 else if (args[i].equals("-s") && !verify)
  signername = args[++i];
 else if (args[i].equals("-sa") && !verify)
  signatureAlgorithm = args[++i];
 else if (args[i].equals("-p"))
  password = args[++i];
 else if (args[i].equals("-keystore"))
  keystoreFile = new File(args[++i]);
 else if (args[i].equals("-keystoreType"))
  keystoreType = args[++i];
 else if (args[i].equals("-keystorePassword"))
  keystorePassword = args[++i];

 else if (!verify) filelist.add(args[i]);
 else throw new IllegalArgumentException(args[i]);
}

// Если некоторые аргументы не заданы, даем им принимаемые
// по умолчанию значения.
if (keystoreFile == null) {
 File dir = new File(System.getProperty("user.home"));
 keystoreFile = new File(dir, ".keystore");
}
if (keystoreType == null) keystoreType = KeyStore.getDefaultType();
if (keystorePassword == null) keystorePassword = password;

if (!verify && signername != null && password == null) {
 System.out.println("Используйте -p для задания пароля.");
 return;
}

// Получаем хранилище, которое будем использовать для подписания и проверки
// подписей. Если пароль не задан, полагаем, что нам не придется иметь дела
// с подписями, и пропускаем хранилище.
KeyStore keystore = null;
if (keystorePassword != null) {
 keystore = KeyStore.getInstance(keystoreType);
 InputStream in =
  new BufferedInputStream(new FileInputStream(keystoreFile));
 keystore.load(in, keystorePassword.toCharArray());
}

// Если задана опция -v или не задано никаких файлов, проверяем манифест.
// В противном случае создаем новый манифест для заданных файлов
if (verify || (filelist.size() == 0)) verify(manifestfile, keystore);
else create(manifestfile, digestAlgorithm,
```

```
    signername, signatureAlgorithm,
    keystore, password, filelist);
}

/**
 * Этот метод создает файл манифеста с заданным именем для заданного вектора
 * файлов, используя указанный алгоритм дайджестов сообщений.
 * Если signername имеет значение, отличное от null, этот метод добавляет
 * в манифест цифровую подпись с использованием названного алгоритма
 * создания подписи. Этот метод может выдать множество исключений.
 **/
public static void create(String manifestfile, String digestAlgorithm,
       String signername, String signatureAlgorithm,
       KeyStore keystore, String password,
       List filelist)
 throws NoSuchAlgorithmException, InvalidKeyException,
   SignatureException, KeyStoreException,
   UnrecoverableKeyException, IOException
{
 // Для вычисления подписи мы должны обрабатывать файлы в фиксированном,
 // воспроизводимом порядке, так что отсортируем их по алфавиту.
 Collections.sort(filelist);
 int numfiles = filelist.size();

 Properties manifest = new Properties(), metadata = new Properties();
 MessageDigest md = MessageDigest.getInstance(digestAlgorithm);
 Signature signature = null;
 byte[] digest;

 // Если имя генератора подписей задано, подготовимся к подписанию манифеста
 if (signername != null) {
  // Получаем объект Signature
  signature = Signature.getInstance(signatureAlgorithm);

  // Ищем закрытый ключ генератора в хранилище
  PrivateKey key = (PrivateKey)
   keystore.getKey(signername, password.toCharArray());

  // Теперь готовимся создать подпись для заданного генератора
  signature.initSign(key);
 }

 // Теперь перебираем файлы в цикле в алфавитном порядке,
 // что является общепризнанным
 System.out.print("Вычислются дайджесты сообщений");
 for(int i = 0; i < numfiles; i++) {
  String filename = (String)filelist.get(i);
  // Вычисляем дайджест для каждого существующего файла,
  // перескакивая через несуществующие.
  try { digest = getFileDigest(filename, md); }
  catch (IOException e) {
   System.err.println("\nПропускаем " + filename + ": " + e);
   continue;
  }
```

```
        // Если вычисляем подпись, используем байты из имени файла
        // и дайджеста в качестве части подписываемых данных.
        if (signature != null) {
          signature.update(filename.getBytes());
          signature.update(digest);
        }
        // Сохраняем в манифесте имя файла и закодированные байты дайджеста
        manifest.put(filename, hexEncode(digest));
        System.out.print('.');
        System.out.flush();
      }

      // Если генератор подписей задан, вычисляем подпись для манифеста
      byte[] signaturebytes = null;
      if (signature != null) {
        System.out.print("готово\nВычисляем цифровую подпись...");
        System.out.flush();

        // С использованием закрытого ключа генератора вычисляем цифровую подпись
        // путем вычисления дайджеста сообщения для всех байтов, переданных
        // методу update(). Это потребует некоторого времени.
        signaturebytes = signature.sign();
      }

      // Сообщаем пользователю, что происходит
      System.out.print("готово\nЗаписываем манифест...");
      System.out.flush();

      // Сохраняем некоторые метаданные о манифесте, включая сюда имя
      // используемого алгоритма создания дайджеста сообщения
      metadata.put("__META.DIGESTALGORITHM", digestAlgorithm);
      // Если манифест подписывается, сохраняем еще кое-какие метаданные
      if (signername != null) {
        // Сохраняем имя генератора подписей
        metadata.put("__META.SIGNER", signername);
        // Сохраняем имя алгоритма
        metadata.put("__META.SIGNATUREALGORITHM", signatureAlgorithm);
        // Генерируем подпись, кодируем ее и сохраняем
        metadata.put("__META.SIGNATURE", hexEncode(signaturebytes));
      }

      // Теперь сохраняем данные манифеста и метаданные в файле манифеста
      FileOutputStream f = new FileOutputStream(manifestfile);
      manifest.store(f, "Дайджесты сообщений манифеста");
      metadata.store(f, "Метаданные манифеста");
      System.out.println("Готово");
    }

    /**
     * Этот метод проверяет цифровую подпись (если она имеется) заданного файла
     * манифеста, и если проверка проходит успешно, проверяет дайджесты
     * сообщения для каждого файла из списка файлов, имя которого также
     * содержится в манифесте. Этот метод может выдавать множество исключений
     **/
```

```
public static void verify(String manifestfile, KeyStore keystore)
 throws NoSuchAlgorithmException, SignatureException,
   InvalidKeyException, KeyStoreException, IOException
{
 Properties manifest = new Properties();
 manifest.load(new FileInputStream(manifestfile));
 String digestAlgorithm =
  manifest.getProperty("__META.DIGESTALGORITHM");
 String signername = manifest.getProperty("__META.SIGNER");
 String signatureAlgorithm =
  manifest.getProperty("__META.SIGNATUREALGORITHM");
 String hexsignature = manifest.getProperty("__META.SIGNATURE");

 // Получаем из манифеста список файлов.
 List files = new ArrayList();
 Enumeration names = manifest.propertyNames();
 while(names.hasMoreElements()) {
  String s = (String)names.nextElement();
  if (!s.startsWith("__META")) files.add(s);
 }
 int numfiles = files.size();

 // Если манифест подписан, а хранилище ключей не задано,
 // предупреждаем пользователя
 if (signername != null && keystore == null)
  System.out.println("Невозможна проверка цифровой подписи без " +
       "хранилища ключей.");

 // Если манифест содержит метаданные, касающиеся цифровой подписи,
 // первым делом проверяем эту подпись
 if (signername != null && keystore != null) {
  System.out.print("Проверка цифровой подписи...");
  System.out.flush();

  // Для проверки подписи мы должны обработать файлы точно в том порядке,
  // в каком они обрабатывались при создании подписи. Мы
  // обеспечиваем этот порядок за счет сортировки файлов.
  Collections.sort(files);

  // Создаем объект Signature для проверки подписи.
  // Инициализируем его при помощи открытого ключа генератора подписей,
  // взятого из хранилища ключей
  Signature signature = Signature.getInstance(signatureAlgorithm);
  PublicKey publickey =
   keystore.getCertificate(signername).getPublicKey();
  signature.initVerify(publickey);

  // Теперь в цикле перебираем файлы в порядке, заданном сортировкой.
  // Для каждого из них посылаем байты имени файла и дайджеста
  // объекту Signature, где они будут использованы для вычисления подписи.
  // Важно, чтобы при проверке подписи это делалось точно в том же порядке,
  // что и при ее создании

  for(int i = 0; i < numfiles; i++) {
```

```
    String filename = (String) files.get(i);
    signature.update(filename.getBytes());
    signature.update(hexDecode(manifest.getProperty(filename)));
   }

   // Теперь расшифровываем подпись, считанную из файла манифеста,
   // и передаем ее методу verify() объекта Signature. Если подпись
   // не подтверждается, печатаем сообщение об ошибке и выходим.
   if (!signature.verify(hexDecode(hexsignature))) {
    System.out.println("\nПодпись манифеста недействительна");
    System.exit(0);
   }

   // Сообщаем пользователю, что эти долгие вычисления окончены
   System.out.println("подтверждена.");
  }

  // Сообщаем пользователю, что переходим к следующей фазе проверки
  System.out.print("Проверяем дайджесты сообщений файлов");
  System.out.flush();

  // Получаем объект MessageDigest для вычисления дайджестов
  MessageDigest md = MessageDigest.getInstance(digestAlgorithm);
  // Цикл по всем файлам
  for(int i = 0; i < numfiles; i++) {
   String filename = (String)files.get(i);
   // Ищем закодированный дайджест в файле манифеста
   String hexdigest = manifest.getProperty(filename);
   // Вычисляем дайджест для файла.
   byte[] digest;
   try { digest = getFileDigest(filename, md); }
   catch (IOException e) {
    System.out.println("\nПропускаем " + filename + ": " + e);
    continue;
   }

   // Кодируем вычисленный дайджест и сравниваем его с закодированным
   // дайджестом из манифеста. Если они не совпадают, печатаем
   // сообщение об ошибке.
   if (!hexdigest.equals(hexEncode(digest)))
    System.out.println("\nФайл '" + filename +
         "' не прошел проверку.");

   // Выводим по точке для каждого обработанного файла. Поскольку вычисление
   // дайджестов сообщения занимает некоторое время, это покажет
   // пользователю, что программа работает и процесс идет
   System.out.print(".");
   System.out.flush();
  }
  // Наконец, сообщаем пользователю, что проверка окончена.
  System.out.println("готово.");
 }

 /**
```

```
 * Этот вспомогательный метод используется обоими методами create()
 * и verify(). Он считывает содержимое заданного файла и вычисляет для него
 * дайджест сообщения, используя заданный объект MessageDigest.
 **/
public static byte[] getFileDigest(String filename, MessageDigest md)
 throws IOException {
 // Очищаем объект MessageDigest
 md.reset();

 // Создаем поток для чтения из файла и вычисляем дайджест
 DigestInputStream in =
  new DigestInputStream(new FileInputStream(filename),md);

 // Читаем файл до конца, отбрасывая все прочитанное.
 // Поток DigestInputStream автоматически передает все считанные байты
 // методу update() объекта MessageDigest
 while(in.read(buffer) != -1) /* не делаем ничего */ ;

 // Наконец, вычисляем и возвращаем значение дайджеста.
 return md.digest();
}

/** Этот статический буфер используется вышеприведенным
    методом getFileDigest()*/
public static byte[] buffer = new byte[4096];

/** Этот массив используется при преобразовании байтов
    в шестнадцатеричные числа */
static final char[] digits = { '0', '1', '2', '3', '4', '5', '6', '7',
                               '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'};

/**
 * Вспомогательный метод для преобразования массива байтов в объект String.
 * Мы делаем это, просто преобразуя каждый байт в две шестнадцатеричные
 * цифры. Кодировка наподобие Base 64 была бы более компактной,
 * но более трудной для вычисления.
 **/
public static String hexEncode(byte[] bytes) {
 StringBuffer s = new StringBuffer(bytes.length * 2);
 for(int i = 0; i < bytes.length; i++) {
  byte b = bytes[i];
  s.append(digits[(b& 0xf0) >> 4]);
  s.append(digits[b& 0x0f]);
 }
 return s.toString();
}

/**
 * Вспомогательный метод для преобразования в обратном направлении –
 * из строки шестнадцатеричных цифр в массив байтов.
 **/
public static byte[] hexDecode(String s) throws IllegalArgumentException {
 try {
  int len = s.length();
```

```
    byte[] r = new byte[len/2];
    for(int i = 0; i < r.length; i++) {
     int digit1 = s.charAt(i*2), digit2 = s.charAt(i*2 + 1);
     if ((digit1 >= '0') && (digit1 <= '9')) digit1 -= '0';
     else if ((digit1 >= 'a')&& (digit1 <= 'f')) digit1 -= 'a' - 10;
     if ((digit2 >= '0') && (digit2 <= '9')) digit2 -= '0';
     else if ((digit2 >= 'a')&& (digit2 <= 'f')) digit2 -= 'a' - 10;
     r[i] = (byte)((digit1 << 4) + digit2);
    }
    return r;
   }
   catch (Exception e) {
    throw new IllegalArgumentException("hexDecode(): неверный ввод");
   }
  }
 }
```

Криптография

Ядро платформы Java не содержит поддержку шифрования и дешифрования данных в силу строгих правил экспорта США. Однако эти технологии поддерживает пакет Java Cryptography Extension, или JCE. Чтобы воспользоваться ими, нужно только скачать JCE с *http://java.sun.com/products/jce/* и установить его. Обратите внимание на то, что JCE 1.2.1 (в бета-версии на момент написания этой книги) доступен в глобально экспортируемой версии, поддерживающей только слабое шифрование с ключами уменьшенного размера. Если вы находитесь за пределами Соединенных Штатов и Канады, вы можете пользоваться этой версией JCE или приобрести другую реализацию, разработанную не в США и, следовательно, не подверженную ограничительным правилам экспорта.

Чтобы установить JCE, просто скопируйте все входящие в него файлы JAR в каталог *jre/lib/ext/* вашего дистрибутива Java. Затем, чтобы сделать алгоритмы JCE автоматически доступными для всех Java-программ, отредактируйте файл *jre/lib/security/java.security*, включив в него следующую строку:

```
security.provider.3=com.sun.crypto.provider.SunJCE
```

Прочитайте комментарии в файле *java.security* для получения дополнительной информации о том, что эта строка делает.

Пример 6.5 является программой, позволяющей шифровать/дешифровать файлы с применением шифровального алгоритма TripleDES и генерировать ключи TripleDES, сохраняемые в файлах. Она использует классы JCE из пакета `javax.crypto` и его подпакетов. Главными классами здесь являются класс `Cipher`, представляющий алгоритм шифрования/дешифрования, и класс `SecretKey`, представляющий используе-

мые алгоритмом ключи шифрования и дешифрования. Справочные сведения по API для классов JCE можно найти в книге «Java in a Nutshell». Дополнительные сведения о криптографии и JCE можно почерпнуть в книге Джонатана Нудсена (Jonathan Knudsen) «Java Cryptography» (издательство O'Reilly).

Пример 6.5. TripleDES.java

```
package com.davidflanagan.examples.security;
import javax.crypto.*;
import javax.crypto.spec.*;
import java.security.*;
import java.security.spec.*;
import java.io.*;

/**
 * Этот класс определяет методы для шифрования/дешифрования с использованием
 * алгоритма TripleDES и для генерирования, чтения и записи ключей Triple DES.
 * В нем определяется также метод main(), позволяющий вызывать эти методы
 * из командной строки.
 **/
public class TripleDES {
 /**
  * Программа. Первый аргумент должен быть флагом -e, -d или -g
  * для шифрования, дешифрования или генерирования ключа. Второй аргумент -
  * имя файла, из которого ключ считывается или в который записывается
  * (при флаге -g). Аргументы -e и -d заставляют программу читать
  * из стандартного входного потока и направлять шифрованный
  * или дешифрованный текст в стандартный выходной поток.
  **/
 public static void main(String[] args) {
  try {
   // Проверяем, существует ли провайдер услуги по шифрованию
   // методом TripleDES. Если нет, явным образом устанавливаем
   // в этом качестве SunJCE.
   try { Cipher c = Cipher.getInstance("DESede"); }
   catch(Exception e) {
    // Возникшее здесь исключение свидетельствует, вероятно, о том,
    // что провайдер JCE в этой системе не был установлен на постоянной
    // основе, то есть не был включен
    // в файл $JAVA_HOME/jre/lib/security/java.security.
    // Значит, нам нужно явным образом установить провайдер JCE.
    System.err.println("Установка провайдера SunJCE.");
    Provider sunjce = new com.sun.crypto.provider.SunJCE();
    Security.addProvider(sunjce);
   }

   // Это то место, из которого мы будем считывать ключ или в которое
   // будем его записывать
   File keyfile = new File(args[1]);

   // Теперь смотрим на первый аргумент, чтобы узнать, что нам следует делать
   if (args[0].equals("-g")) {  // Генерировать ключ
```

```
   System.out.print("Генерация ключа. Это может занять некоторое время...");
   System.out.flush();
   SecretKey key = generateKey();
   writeKey(key, keyfile);
   System.out.println("готово.");
   System.out.println("Секретный ключ записан в " + args[1] +
        ". Никому не показывайте этот файл!");
  }
  else if (args[0].equals("-e")) { // Шифровать стандартный входной
                                   // поток в стандартный выходной
   SecretKey key = readKey(keyfile);
   encrypt(key, System.in, System.out);
  }
  else if (args[0].equals("-d")) { // Дешифровать стандартный входной
                                   // поток в стандартный выходной
   SecretKey key = readKey(keyfile);
   decrypt(key, System.in, System.out);
  }
 }
 catch(Exception e) {
  System.err.println(e);
  System.err.println("Формат: java " + TripleDES.class.getName() +
       " -d|-e|-g <keyfile>");
 }
}

/** Генерируем секретный ключ TripleDES для шифрования/дешифрования */
public static SecretKey generateKey() throws NoSuchAlgorithmException {
 // Получаем генератор ключей от Triple DES (известный как DESede)
 KeyGenerator keygen = KeyGenerator.getInstance("DESede");
 // Применяем его для генерации ключа
 return keygen.generateKey();
}

/** Сохраняем заданный TripleDES SecretKey в заданном файле */
public static void writeKey(SecretKey key, File f)
 throws IOException, NoSuchAlgorithmException, InvalidKeySpecException
{
 // Преобразуем секретный ключ в массив байтов, например, так
 SecretKeyFactory keyfactory = SecretKeyFactory.getInstance("DESede");
 DESedeKeySpec keyspec =
  (DESedeKeySpec)keyfactory.getKeySpec(key, DESedeKeySpec.class);
 byte[] rawkey = keyspec.getKey();

 // Записываем необработанный ключ в файл
 FileOutputStream out = new FileOutputStream(f);
 out.write(rawkey);
 out.close();
}

/** Читаем секретный ключ TripleDES из заданного файла */
public static SecretKey readKey(File f)
 throws IOException, NoSuchAlgorithmException,
```

```
    InvalidKeyException, InvalidKeySpecException
{
 // Читаем необработанные байты из keyfile
 DataInputStream in = new DataInputStream(new FileInputStream(f));
 byte[] rawkey = new byte[(int)f.length()];
 in.readFully(rawkey);
 in.close();

 // Преобразуем необработанные байты в секретный ключ примерно так
 DESedeKeySpec keyspec = new DESedeKeySpec(rawkey);
 SecretKeyFactory keyfactory = SecretKeyFactory.getInstance("DESede");
 SecretKey key = keyfactory.generateSecret(keyspec);
 return key;
}

/**
 * Применяем заданный ключ TripleDES для шифрования байтов из потока ввода
 * и записываем их в поток вывода. Этот метод использует CipherOutputStream
 * для шифрования и одновременной записи байтов.
 **/
public static void encrypt(SecretKey key, InputStream in, OutputStream out)
 throws NoSuchAlgorithmException, InvalidKeyException,
   NoSuchPaddingException, IOException
{
 // Создаем и инициализируем шифровальную машинку
 Cipher cipher = Cipher.getInstance("DESede");
 cipher.init(Cipher.ENCRYPT_MODE, key);

 // Создаем специальный поток вывода, который проделает работу за нас
 CipherOutputStream cos = new CipherOutputStream(out, cipher);

 // Читаем ввод и пишем в шифрующий поток вывода
 byte[] buffer = new byte[2048];
 int bytesRead;
 while((bytesRead = in.read(buffer)) != -1) {
  cos.write(buffer, 0, bytesRead);
 }
 cos.close();

 // В целях большей безопасности не оставляем открытый текст
 // болтаться в памяти.
 java.util.Arrays.fill(buffer, (byte) 0);
}

/**
 * Применяем заданный ключ TripleDES для дешифрования байтов из потока ввода
 * и записи их в поток вывода. Этот метод использует класс Cipher напрямую,
 * чтобы показать, как можно обойтись без потоков CipherInputStream
 * и CipherOutputStream.
 **/
public static void decrypt(SecretKey key, InputStream in, OutputStream out)
 throws NoSuchAlgorithmException, InvalidKeyException, IOException,
   IllegalBlockSizeException, NoSuchPaddingException,
   BadPaddingException
```

```
    {
     // Создаем и инициализируем дешифровальную машинку
     Cipher cipher = Cipher.getInstance("DESede");
     cipher.init(Cipher.DECRYPT_MODE, key);

     // Читаем байты, дешифруем и пишем их в выходной поток.
     byte[] buffer = new byte[2048];
     int bytesRead;
     while((bytesRead = in.read(buffer)) != -1) {
      out.write(cipher.update(buffer, 0, bytesRead));
     }

     // Дописываем заключительную порцию расшифрованных байтов
     out.write(cipher.doFinal());
     out.flush();
    }
   }
```

Упражнения

6-1. Напишите класс `PasswordManager`, связывающий имена пользователей с паролями и обладающий методами, которые создают и удаляют пары «имя пользователя/пароль», изменяют пароль, связанный с именем пользователя, и проводят аутентификацию пользователя путем проверки предъявленного пароля. Класс `PasswordManager` должен сохранять имена пользователей и пароли в файле (или в базе данных, если вы уже прочитали главу 17).

Заметьте, однако, что класс не должен хранить пароли в виде простого текста, что позволило бы незваному гостю, взломавшему систему `PasswordManager`, получить полный доступ ко всем паролям. Чтобы предупредить такую возможность, обычно используют функцию невосстановимого шифрования паролей. Дайджесты сообщения, как те, что использовались в примере 6.2, демонстрируют именно такого рода однонаправленную функцию. Вычислить дайджест сообщения по паролю относительно просто, но восстановить пароль из дайджеста очень трудно, а то и невозможно.

Сконструируйте класс `PasswordManager` так, чтобы вместо хранения действительного пароля он сохранял только дайджест сообщения для пароля. Для проверки пользовательского пароля класс должен вычислять дайджест для предъявленного пароля и сравнивать его с хранимым дайджестом. Если дайджесты совпали, можно считать, что пароли тоже совпадают. (В действительности есть бесконечно малая вероятность того, что разные пароли породят одинаковые дайджесты сообщения, но этой возможностью вы можете пренебречь.)

6-2. Напишите сетевую службу и клиентскую программу, позволяющие пользователю изменять пароль, зарегистрированный вашим

классом PasswordManager. Если вы уже прочитали главу 16 «Вызов удаленных методов», преобразуйте PasswordManager так, чтобы он запускался как удаленный объект RMI, и напишите клиентскую программу, использующую этот удаленный объект для изменения пароля. Если вы еще не прочитали эту главу, напишите службу изменения пароля, работающую под управлением класса Server, разработанного в главе 5, и примените класс GenericClient из той же главы для взаимодействия с этой службой. В любом случае создайте файл политик с набором прав доступа, требуемых вашей сетевой службой, и используйте этот файл политик, чтобы обеспечить возможность запуска вашей службы с опцией интерпретатора Java -Djava.security.manager.

6-3. Класс TripleDES из примера 6.5 использует алгоритм DESede в принимаемом по умолчанию режиме ECB (electronic code book). Этот режим шифрования более уязвим для некоторых попыток дешифрования, чем режим CBC (cipher block chaining). Преобразуйте пример так, чтобы он использовал режим CBC. Режим задается как часть имени алгоритма: в этом случае вместо «DESede» в качестве алгоритма следует задавать «DESede/CBC/PKCS5Padding».

Для шифрования в режиме CBC объект Cipher создает инициализирующий вектор (initialization vector, IV) случайных байтов, который понадобится и при дешифровании. Преобразуйте метод encrypt() так, чтобы он получал IV при помощи метода getIV() объекта Cipher и записывал байты (и длину) массива IV в поток вывода перед записью зашифрованных байтов. Для этой цели вы, возможно, захотите переделать метод encrypt() так, чтобы он не использовал поток CipherOutputStreamx, а работал с классом Cipher напрямую, как это делает метод decrypt(). Преобразуйте метод decrypt() так, чтобы он читал байты вектора IV и использовал их для создания объекта javax.crypto.spec.IvParameterSpec, который он затем передает (в виде AlgorithmParameterSpec) одному из методов init() объекта Cipher.

6-4. Программа TripleDES сохраняет секретные ключи в незащищенных файлах, а это не самый безопасный способ работы с важными ключами. Преобразуйте программу так, чтобы она использовала объект KeyStore для сохранения (и получения) ключей в защищенном паролем виде. Класс KeyStore был продемонстрирован в примере 6.2, где он использовался для хранения объектов PublicKey и PrivateKey при создании цифровых подписей. Однако KeyStore может также хранить объекты SecretKey. Просто передайте SecretKey методу setKeyEntry(), задав имя для этого пароля и пароль для его защиты. Поскольку этот ключ не является объектом класса PrivateKey, аргументу Certificate[] этого метода следует присвоить значение null.

Глава 7

Интернационализация

Интернационализация – это процесс придания программе гибкости, достаточной, чтобы она корректно работала при любых местных установках. Необходимым дополнением к интернационализации является *локализация* – процесс настройки программы для работы при определенных местных установках.

Задача интернационализации распадается на несколько отличающихся друг от друга шагов. Java (1.1 и более поздних версий) для выполнения этих шагов применяет разные механизмы:

- Программа должна уметь читать, записывать и обрабатывать локализованный текст. Java применяет кодировку символов Unicode, что само по себе является большим шагом по направлению к интернационализации. Кроме того, классы `InputStreamReader` и `OutputStreamWriter` преобразуют текст соответственно из локальных кодировок в Unicode и из Unicode в локальные кодировки.
- Программа должна соответствовать местным правилам отображения даты и времени дня, форматирования чисел и сортировки строк. По этим вопросам Java отсылает нас к классам пакета `java.text`.
- Программа должна отображать весь видимый для пользователя текст на его родном языке. Перевод выводимых программой сообщений – всегда одна из главных задач при локализации программы. Еще более важной задачей является такое написание программы, чтобы весь видимый для пользователя текст она получала во время выполнения, а не содержала в своем коде. Java облегчает этот процесс при помощи класса `ResourceBundle` и его подклассов, содержащихся в пакете `java.util`.

В этой главе обсуждаются все три аспекта интернационализации.

Несколько слов о регионах

Под английским термином *locale* понимается определенный географический, политический или культурный регион. В Java регионы представлены классом `java.util.Locale`. Регионы часто определяются языком, который обозначается своим стандартным двухбуквенным кодом в нижнем регистре, например en (английский) или fr (французский). Однако иногда, чтобы однозначно указать конкретный регион, только языка недостаточно, и в спецификацию добавляется код страны. Страна представляется двухбуквенным кодом в верхнем регистре. Например, регион, соответствующий американскому диалекту английского языка (en_US), отличается от региона британского английского языка (en_GB), а французский, на котором говорят в Канаде (fr_CA), отличается от французского, на котором говорят во Франции (fr_FR). Иногда границы региона еще более сужаются путем задания зависящей от системы строки варианта.

Класс `Locale` поддерживает статический регион по умолчанию, который может устанавливаться и опрашиваться методами `Locale.setDefault()` и `Locale.getDefault()` соответственно. В Java чувствительные к региону методы обычно принадлежат одному из двух видов. Первые используют регион, используемый по умолчанию, а другие используют объект `Locale`, явно заданный в качестве их аргумента. Программа может создавать и использовать любое число отличных от принимаемых по умолчанию объектов `Locale`, хотя чаще просто полагаются на используемый по умолчанию регион, который наследуется от региона по умолчанию «родной» платформы. В Java чувствительные к регионам классы часто предоставляют метод, позволяющий запросить список поддерживаемых ими регионов.

Наконец, обратите внимание на то, что у компонентов AWT и Swing GUI (см. главу 10) есть свойство `locale`, что делает возможным применение различных регионов для разных компонентов. (Большинство компонентов, однако, нечувствительны к регионам; они ведут себя одинаково в любом регионе.)

Кодировка Unicode

Java использует кодировку символов Unicode. (Java 1.3 использует Unicode Version 2.1. Поддержка для Unicode 3.0 будет включена в Java 1.4 или другую будущую версию.) Unicode – это 16-битовая кодировка символов, созданная консорциумом Unicode Consortium, который описывает стандарт следующим образом (см. *http://unicode.org*):

> Стандарт Unicode определяет коды для символов, используемых в большинстве современных письменных языков. Шрифты включают в себя европейские алфавиты, ближневосточные шрифты для письма

> справа налево и шрифты Азии. Стандарт Unicode включает в себя знаки препинания, диакритические знаки, математические символы, технические символы, стрелки, псевдографику и т. д. ...В совокупности стандарт Unicode дает коды для 49 194 символов мировых алфавитов, идеографических наборов и коллекций символов.

В канонической форме кодировки Unicode, используемой в Java типами данных `char` и `String`, каждый символ занимает два байта. Символы Unicode от `\u0020` до `\u007E` эквивалентны символам ASCII и ISO8859-1 (Latin-1) с кодами от `0x20` до `0x7E`. Символы Unicode от `\u00A0` до `\u00FF` идентичны символам ISO8859-1 от `0xA0` до `0xFF`. Таким образом, существует тривиальное соответствие между символами Latin-1 и символами Unicode. Множество других фрагментов кодировки Unicode основаны на существующих стандартах, например ISO8859-5 (кириллица) и ISO8859-8 (еврейский алфавит), хотя соответствие между этими стандартами и Unicode, возможно, не так тривиально, как в случае Latin-1.

Обратите внимание на то, что поддержка Unicode на многих платформах может быть ограниченной. Одна из трудностей при использовании Unicode состоит в недостатке доступных шрифтов для отображения всех символов Unicode. На рис. 7.1 показаны некоторые символы стандартных шрифтов, поставляемых компанией Sun в составе Java 1.3 SDK для Linux. (Обратите внимание на то, что эти шрифты не поставляются с Java JRE, так что даже если они доступны на платформе разработчика, они, возможно, окажутся недоступны на той платформе, где будут исполняться приложение или апплет.) Обратите внимание на специальный символ (пустой квадратик), которым отмечаются неопределенные символы.

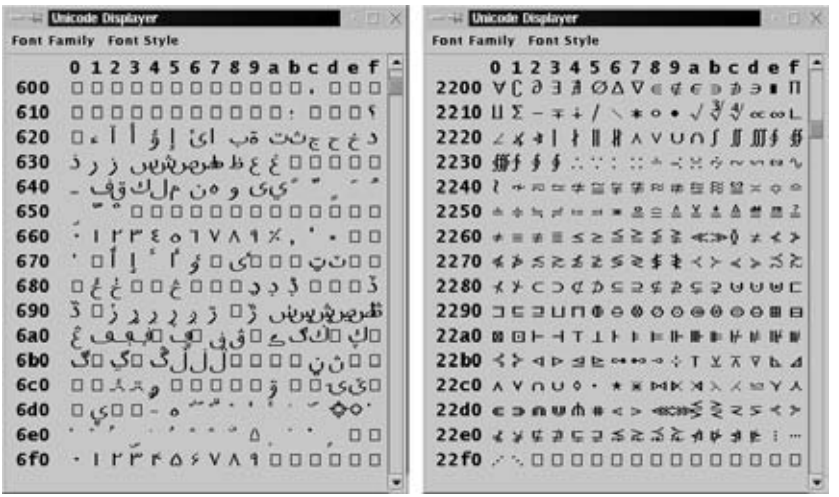


Рис. 7.1. *Некоторые символы Unicode и их коды*

Пример 7.1 содержит коды, использованные для создания рис. 7.1. Поскольку символы Unicode так фундаментально встроены в язык Java, эта программа для отображения символов Unicode `UnicodeDisplay` не предпринимает никаких изощренных действий по интернационализации. Вы увидите, что пример 7.1 скорее подходит в качестве примера применения Swing GUI, чем примера интернационализации. Если вы еще не прочитали главу 10, то, возможно, поймете не весь код этого примера.

Пример 7.1. UnicodeDisplay.java

```
package com.davidflanagan.examples.i18n;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
 * Эта программа отображает символы Unicode с применением заданных
 * пользователем шрифтов и стилей.
 **/
public class UnicodeDisplay extends JFrame implements ActionListener {
  int page = 0;
  UnicodePanel p;
  JScrollBar b;
  String fontfamily = "Serif";
  int fontstyle = Font.PLAIN;

  /**
   * Этот конструктор создает рамку, полосу меню и полосу прокрутки,
   * работающие совместно с определенным ниже классом UnicodePanel
   **/
  public UnicodeDisplay(String name) {
    super(name);
    p = new UnicodePanel();              // Создаем панель
    p.setBase((char)(page * 0x100));     // Инициализируем ее
    getContentPane().add(p, "Center");   // Центрируем ее

    // Создаем и настраиваем полосу прокрутки и помещаем ее справа
    b = new JScrollBar(Scrollbar.VERTICAL, 0, 1, 0, 0xFF);
    b.setUnitIncrement(1);
    b.setBlockIncrement(0x10);
    b.addAdjustmentListener(new AdjustmentListener() {
        public void adjustmentValueChanged(AdjustmentEvent e) {
          page = e.getValue();
          p.setBase((char)(page * 0x100));
        }
      });
    getContentPane().add(b, "East");

    // Задаем реакцию на требование закрыть окно
    this.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) { System.exit(0); }
      });
```

```
    // Обрабатываем нажатия клавиш Page Up, Page Down и клавиш
    // со стрелками вверх и вниз
    this.addKeyListener(new KeyAdapter() {
        public void keyPressed(KeyEvent e) {
          int code = e.getKeyCode();
          int oldpage = page;
          if ((code == KeyEvent.VK_PAGE_UP) ||
            (code == KeyEvent.VK_UP)) {
            if (e.isShiftDown()) page -= 0x10;
            else page -= 1;
            if (page < 0) page = 0;
          }
          else if ((code == KeyEvent.VK_PAGE_DOWN) ||
              (code == KeyEvent.VK_DOWN)) {
            if (e.isShiftDown()) page += 0x10;
            else page += 1;
            if (page > 0xff) page = 0xff;
          }
          if (page != oldpage) {   // если что-то переменилось...
            p.setBase((char) (page * 0x100)); // обновляем отображение
            b.setValue(page);       // и соответственно обновляем
                                    // полосу прокрутки
          }
        }
      });

    // Устанавливаем меню изменения шрифтов.
    // Используем вспомогательный метод.
    JMenuBar menubar = new JMenuBar();
    this.setJMenuBar(menubar);
    menubar.add(makemenu("Семейство шрифтов",
                new String[] {"Serif", "SansSerif", "Monospaced"},
                this));
    menubar.add(makemenu("Стиль шрифта",
                new String[]{
                  "Plain","Italic","Bold","BoldItalic"
                }, this));
  }

  /** Этот метод обрабатывает элементы меню */
  public void actionPerformed(ActionEvent e) {
    String cmd = e.getActionCommand();
    if (cmd.equals("Serif")) fontfamily = "Serif";
    else if (cmd.equals("SansSerif")) fontfamily = "SansSerif";
    else if (cmd.equals("Monospaced")) fontfamily = "Monospaced";
    else if (cmd.equals("Plain")) fontstyle = Font.PLAIN;
    else if (cmd.equals("Italic")) fontstyle = Font.ITALIC;
    else if (cmd.equals("Bold")) fontstyle = Font.BOLD;
    else if (cmd.equals("BoldItalic")) fontstyle = Font.BOLD + Font.ITALIC;
    p.setFont(fontfamily, fontstyle);
  }
```

```
/** Вспомогательный метод, создающий меню из массива элементов */
private JMenu makemenu(String name, String[] itemnames,
            ActionListener listener)
{
  JMenu m = new JMenu(name);
  for(int i = 0; i < itemnames.length; i++) {
    JMenuItem item = new JMenuItem(itemnames[i]);
    item.addActionListener(listener);
    item.setActionCommand(itemnames[i]); // вот и все
    m.add(item);
  }
  return m;
}

/** Программа main() просто создает окно, пакует его и показывает его */
public static void main(String[] args) {
  UnicodeDisplay f = new UnicodeDisplay("Unicode Displayer");
  f.pack();
  f.show();
}

/**
 * Этот вложенный класс отображает одну «страницу» символов Unicode
 * за один раз. Каждая «страница» представляет 256 символов, расположенных
 * в 16 строках и 16 столбцах.
 **/
public static class UnicodePanel extends JComponent {
  protected char base; // С какого символа мы начинаем
  protected Font font = new Font("serif", Font.PLAIN, 18);
  protected Font headingfont = new Font("monospaced", Font.BOLD, 18);
  static final int lineheight = 25;
  static final int charspacing = 20;
  static final int x0 = 65;
  static final int y0 = 40;

  /** Указываем, с какого места начинаем вывод, и перерисовываем */
  public void setBase(char base) { this.base = base; repaint(); }

  /** Устанавливаем новый шрифт или стиль и перерисовываем */
  public void setFont(String family, int style) {
    this.font = new Font(family, style, 18);
    repaint();
  }

  /**
   * Метод paintComponent() фактически рисует страницу с символами
   **/
  public void paintComponent(Graphics g) {
    int start = (int)base & 0xFFF0; // Начинаем с символа,
                                    // код которого кратен 16

    // Заголовки выводим специальным шрифтом
    g.setFont(headingfont);
```

```
        // Сверху рисуем цифры 0..F
        for(int i=0; i < 16; i++) {
          String s = Integer.toString(i, 16);
          g.drawString(s, x0 + i*charspacing, y0-20);
        }

        // Слева рисуем столбец.
        for(int i = 0; i < 16; i++) {
          int j = start + i*16;
          String s = Integer.toString(j, 16);
          g.drawString(s, 10, y0+i*lineheight);
        }

        // Теперь рисуем символы
        g.setFont(font);
        char[] c = new char[1];
        for(int i = 0; i < 16; i++) {
          for(int j = 0; j < 16; j++) {
            c[0] = (char)(start + j*16 + i);
            g.drawChars(c, 0, 1, x0 + i*charspacing, y0+j*lineheight);
          }
        }
      }

      /** Специальные компоненты, подобные этому, всегда должны
          содержать этот метод */
      public Dimension getPreferredSize() {
        return new Dimension(x0 + 16*charspacing,
                             y0 + 16*lineheight);
      }
    }
  }
```

Кодировки символов

Представление текста традиционно считалось одной из труднейших проблем при интернационализации. Java, однако, очень элегантно решает эту проблему и скрывает ее трудности. Язык Java использует Unicode неявно, так что он может представить практически любой символ из любого широко распространенного языка. Как я уже говорил, остается еще задача преобразования Unicode в локальные кодировки и обратно. Java включает в себя весьма небольшое число конвертеров «байт-в-символ» и «символ-в-байт», осуществляющих преобразование локальных кодировок символов в Unicode и обратно. Хотя сами по себе эти конвертеры не являются открытыми (`public`), они доступны через классы `InputStreamReader` и `OutputStreamWriter`, символьные потоки, входящие в пакет `java.io`.

Любая программа может автоматически обрабатывать локальные кодировки, просто применяя эти символьные потоки при вводе и выводе

текста. Обратите внимание на то, что классы `FileReader` и `FileWriter` используют эти потоки для автоматического чтения и записи текстовых файлов, использующих кодировки, принятые на данной платформе.

Пример 7.2 показывает простую программу, работающую с кодировками символов. Она преобразует файл из одной заданной кодировки в другую, преобразуя первую кодировку в Unicode, а затем Unicode – во вторую кодировку. Обратите внимание на то, что большую часть программы занимают анализ списка аргументов, обработка исключений и т. п. Нескольких строк оказывается достаточно для создания классов `InputStreamReader` и `OutputStreamWriter`, выполняющих оба преобразования. Также обратите внимание на то, что исключения обрабатываются вызовом `LocalizedError.display()`. Этот метод не является частью Java API; это специальный метод, приведенный в примере 7.7 в конце этой главы.

Пример 7.2. ConvertEncoding.java

```
package com.davidflanagan.examples.i18n;
import java.io.*;

/** Программа преобразования из одной кодировки символов в другую */
public class ConvertEncoding {
  public static void main(String[] args) {
    String from = null, to = null;
    String infile = null, outfile = null;
    for(int i = 0; i < args.length; i++) { // Разбираем аргументы,
                                           // заданные в командной строке.
      if (i == args.length-1) usage();  // Все аргументы требуют еще одного.
      if (args[i].equals("-from")) from = args[++i];
      else if (args[i].equals("-to")) to = args[++i];
      else if (args[i].equals("-in")) infile = args[++i];
      else if (args[i].equals("-out")) outfile = args[++i];
      else usage();
    }

    try { convert(infile, outfile, from, to); } // Попытка преобразования.
    catch (Exception e) {            // Обрабатываем исключения.
      LocalizedError.display(e);     // Определен в конце этой главы.
      System.exit(1);
    }
  }

  public static void usage() {
    System.err.println("Формат: java ConvertEncoding <options>\n" +
               "Ключи:\n\t-from <encoding>\n\t" +
               "-to <encoding>\n\t" +
               "-in <file>\n\t-out <file>");
    System.exit(1);
  }

  public static void convert(String infile, String outfile,
                String from, String to)
      throws IOException, UnsupportedEncodingException
```

```
    {
      // Создаем байтовые потоки.
      InputStream in;
      if (infile != null) in = new FileInputStream(infile);
      else in = System.in;
      OutputStream out;
      if (outfile != null) out = new FileOutputStream(outfile);
      else out = System.out;

      // Если кодировка не задана, используем принятую по умолчанию.
      if (from == null) from = System.getProperty("file.encoding");
      if (to == null) to = System.getProperty("file.encoding");

      // Создаем символьные потоки.
      Reader r = new BufferedReader(new InputStreamReader(in, from));
      Writer w = new BufferedWriter(new OutputStreamWriter(out, to));

      // Копируем символы из потока ввода в поток вывода.
      // Поток InputStreamReader преобразует из входной кодировки в Unicode,
      // а OutputStreamWriter преобразует из Unicode в выходную кодировку.
      // Символы, не имеющие представления в выходной кодировке.
      // отображаются как «?»
  char[] buffer = new char[4096];
      int len;
      while((len = r.read(buffer)) != -1) // Считываем блок ввода.
        w.write(buffer, 0, len);          // И выводим его.
      r.close();              // Закрываем входной поток.
      w.close();              // Проталкиваем и закрываем выходной поток.
    }
  }
```

Учет правил языка

Вторая проблема интернационализации состоит в соблюдении правил языка и традиций в таких областях, как отображение дат и времени дня. Пакет `java.text` определяет классы, облегчающие эту задачу.

Класс `NumberFormat` форматирует числа, денежные суммы и проценты при отображении их для пользователя зависящим от региона способом. Это необходимо, так как в разных регионах есть различные правила форматирования чисел. Во Франции, например, в отличие от англоязычных стран, в качестве отделителя дробной части в десятичных дробях используется запятая, а не точка. Объект `NumberFormat` может применять используемый по умолчанию или любой заданный явно регион.

Класс `DateFormat` форматирует даты и время дня при отображении для пользователя зависящим от региона способом. В разных странах приняты свои правила. Что стоит впереди – день или месяц? Чем отделять часы от минут – точкой или двоеточием? Как на местном языке называются месяцы? Объект `DateFormat` может просто использовать принимаемый по умолчанию или любой заданный вами регион. Класс `Date-`

Format используется в сочетании с классами TimeZone и Calendar пакета java.util. Объект TimeZone сообщает объекту DateFormat, относительно какого часового пояса должна исчисляться дата, а объект Calendar задает способ, по которому сама дата разбивается на дни, недели, месяцы и годы. Почти во всех регионах используется стандартный Григорианский календарь (GregorianCalendar).

Класс Collator сравнивает строки способом, зависящим от региона. Это необходимо, потому что в разных языках строки упорядочиваются по алфавиту разными способами (а в некоторых языках алфавита нет). В традиционном испанском, например, при сортировке буквы «ch» интерпретируются как один символ, расположенный между «c» и «d». При необходимости отсортировать строки или найти строку в тексте Unicode следует использовать объект Collator, созданный для работы с принимаемым по умолчанию или явно заданным регионом.

Класс BreakIterator позволяет обнаружить границы символа, слова, строки и предложения зависящим от региона способом. Он оказывается полезен, когда требуется обнаружить такие границы в тексте Unicode, если, например, вам нужно реализовать алгоритм переноса слов.

Пример 7.3 демонстрирует класс, использующий классы NumberFormat и DateFormat для отображения гипотетического портфеля инвестиций в соответствии с правилами языка. Эта программа использует разные объекты NumberFormat и DateFormat для форматирования (при помощи метода format()) разного рода чисел и дат. Все эти объекты Format действуют с использованием региона по умолчанию, но могли бы быть созданы и с явно заданным регионом. Эта программа отображает сведения о гипотетическом портфеле инвестиций, форматируя даты, числа и денежные суммы в соответствии с текущими или заданными местными установками. Следующий листинг показывает вывод этой программы в соответствии с американскими, британскими и французскими местными установками:

```
% java com.davidflanagan.examples.i18n.Portfolio en US
Стоимость “портфеля”  July 9, 2000 9:08:48 PM PDT:
Symbol  Shares  Purchased      At      Quote   Change
XXX     400     Feb 3, 2000    $11.90  $13.00  9%
YYY     1,100   Mar 2, 2000    $71.09  $27.25  -62%
ZZZ     6,000   May 17, 2000   $23.37  $89.12  281%

% java com.davidflanagan.examples.i18n.Portfolio en GB
Стоимость “портфеля” 09 July 2000 21:08:55 PDT:
Symbol  Shares  Purchased      At      Quote   Change
XXX     400     03-Feb-00      J11.90  J13.00  9%
YYY     1,100   02-Mar-00      J71.09  J27.25  -62%
ZZZ     6,000   17-May-00      J23.37  J89.12  281%

% java com.davidflanagan.examples.i18n.Portfolio fr FR
Стоимость “портфеля” 9 juillet 2000 21:09:03 PDT:
Symbol  Shares  Purchased      At       Quote  Change
```

```
XXX     400     3 fйvr. 00      11,90 F  13,00 F  9%
YYY     1 100   2 mars 00       71,09 F  27,25 F  -62%
ZZZ     6 000   17 mai 00       23,37 F  89,12 F  281%
```

Пример 7.3. Portfolio.java

```
package com.davidflanagan.examples.i18n;
import java.text.*;
import java.util.*;
import java.io.*;

/**
 * Частичная реализация класса гипотетического инвестиционного портфеля.
 * Мы используем его, только чтобы продемонстрировать
 * интернационализацию чисел и дат.
 **/
public class Portfolio {
  EquityPosition[] positions;
  Date lastQuoteTime = new Date();

  public Portfolio(EquityPosition[] positions, Date lastQuoteTime) {
    this.positions = positions;
    this.lastQuoteTime = lastQuoteTime;
  }

  public void print(PrintWriter out) {
    // Получаем объекты NumberFormat и DateFormat для форматирования
    // наших данных.
    NumberFormat number = NumberFormat.getInstance();
    NumberFormat price = NumberFormat.getCurrencyInstance();
    NumberFormat percent = NumberFormat.getPercentInstance();
    DateFormat shortdate = DateFormat.getDateInstance(DateFormat.MEDIUM);
    DateFormat fulldate = DateFormat.getDateTimeInstance(DateFormat.LONG,
                                DateFormat.LONG);

    // Печатаем входные данные.
    out.println("Стоимость "портфеля" " +
          fulldate.format(lastQuoteTime) + ":");
    out.println("Symbol\tShares\tPurchased\tAt\t" +
          "Quote\tChange");

    // Отображаем таблицу, используя методы format() объектов Format.
    for(int i = 0; i < positions.length; i++) {
      out.print(positions[i].name + "\t");
      out.print(number.format(positions[i].shares) + "\t");
      out.print(shortdate.format(positions[i].purchased) + "\t");
      out.print(price.format(positions[i].bought) + "\t");
      out.print(price.format(positions[i].current) + "\t");
      double change =
        (positions[i].current-positions[i].bought)/positions[i].bought;
      out.println(percent.format(change));
      out.flush();
    }
  }
```

```
  static class EquityPosition {
    String name;        // Название ценной бумаги
    int shares;         // Количество наличных бумаг
    Date purchased;     // Дата приобретения
    double bought;      // Цена покупки одной бумаги
    double current;     // Текущая цена одной бумаги
    EquityPosition(String n, int s, Date when, double then, double now) {
      name = n; shares = s; purchased = when;
      bought = then; current = now;
    }
  }

  /**
   * Тестовая программа для демонстрации класса
   **/
  public static void main(String[] args) {
    // Это отображаемый портфель. Заметьте, что для удобства мы используем
    // здесь отмененный конструктор Date(). Он представляет число лет,
    // прошедших после 1900, и при компиляции вызовет выдачу предупреждения.
    EquityPosition[] positions = new EquityPosition[] {
      new EquityPosition("XXX", 400, new Date(100,1,3),11.90,13.00),
      new EquityPosition("YYY", 1100, new Date(100,2,2),71.09,27.25),
      new EquityPosition("ZZZ", 6000, new Date(100,4,17),23.37,89.12)
    };

    // Создаем по этим позициям портфель
    Portfolio portfolio = new Portfolio(positions, new Date());

    // Задаем регион по умолчанию с помощью кодов языка и страны,
    // указанных в командной строке.
    if (args.length == 2) Locale.setDefault(new Locale(args[0], args[1]));

    // Теперь распечатываем портфель
    portfolio.print(new PrintWriter(System.out));
  }
}
```

Установка региона

Пример 7.4 содержит код, где регион задается явным образом с применением кодов языка и страны, указанных в командной строке. Если эти аргументы не заданы, он использует регион, используемый в вашей системе по умолчанию. Экспериментируя с интернационализацией, вы, возможно, захотите изменить принимаемый по умолчанию регион для всей платформы и посмотреть, что из этого выйдет. То, как это делается, зависит от платформы. На Unix-платформах регион назначается обычно путем задания переменной окружения `LANG`. Регион, соответствующий, например, канадскому французскому языку, в Unix-оболочке типа *csh* можно задать командой:

```
% setenv LANG fr_CA
```

А чтобы задать регион для британского английского языка в Unix-оболочке типа *sh*, можно использовать такую команду:

```
$ export LANG=en_GB
```

Для назначения региона в Windows используйте элемент управления **Язык и стандарты** в Панели управления Windows.

Локализация сообщений, выводимых для пользователя

Третья задача интернационализации состоит в разделении кода программы и текста сообщений, адресованных пользователю; содержимое текста должно определяться на основе текущего региона. В примере 7.3, например, строки `"Portfolio value"`, `"Symbol"`, `"Shares"` и другие включены в код приложения и выводятся по-английски, даже если программа будет работать с французским регионом. Единственный способ избежать этого – перевести все сообщения на языки, которые должно поддерживать ваше приложение, и извлекать их в процессе исполнения приложения.

Java помогает справиться с этой задачей при помощи класса `ResourceBundle` пакета `java.util`. Этот класс предоставляет набор ресурсов, которые можно находить по их именам. Для регионов, которые вам нужно поддерживать, следует определить локализованный набор ресурсов, а Java загрузит правильный набор для используемого по умолчанию (или заданного) региона. Когда соответствующий набор загружен, нужные вашей программе ресурсы (обычно это строки) смогут быть найдены в ходе исполнения программы.

Работа с наборами ресурсов

Чтобы определить набор локализованных ресурсов, создается подкласс `ResourceBundle` и даются определения методов `handleGetObject()` и `getKeys()`. Методу `handleGetObject()` передается имя ресурса; он должен возвратить соответствующую локализованную версию ресурса. Метод `getKeys()` должен возвращать объект `Enumeration`, выдающий пользователю список всех имен ресурсов, определенных в `ResourceBundle`. Часто вместо прямого создания подкласса `ResourceBundle` бывает проще создать подкласс `ListResourceBundle`. Также можно просто предоставить файл свойств (см. класс `java.util.Properties`), которым метод `ResourceBundle.getBundle()` воспользуется при создании экземпляра `PropertyResourceBundle`.

Чтобы использовать в программе локализованный ресурс, входящий в набор `ResourceBundle`, следует сначала вызвать статический метод `getBundle()`, который динамически загружает класс `ResourceBundle` и создает его экземпляр. Полученный объект `ResourceBundle` будет иметь за-

данное имя и соответствовать заданному региону (или региону по умолчанию, если он не был задан явно). Получив при помощи `getBundle()` объект `ResourceBundle`, можно применить метод `getObject()` для поиска ресурса по его имени. Обратите внимание на вспомогательный метод `getString()`, преобразующий значение, возвращаемое `getObject()`, в объект `String`.

При вызове `getBundle()` задаются базовое имя требуемого `ResourceBundle` и нужный регион (если вы не хотите использовать регион, принимаемый по умолчанию). Вспомним, что объект `Locale` задается двухбуквенным кодом языка, необязательным двухбуквенным кодом страны и необязательной строкой варианта. Метод `getBundle()` ищет класс `ResourceBundle`, соответствующий текущему региону, путем присоединения информации о регионе к базовому имени набора. Этот метод отыскивает подходящий класс по следующему алгоритму:

1. Ищется класс со следующим именем:

   basename_language_country_variant

   Если такой класс не найден или если для региона не задана строка варианта, переход к следующему шагу.

2. Ищется класс со следующим именем:

   basename_language_country

   Если такой класс не найден или если для региона не задан код страны, переход к следующему шагу.

3. Ищется класс со следующим именем:

   basename_language

   Если такой класс не найден, переход к заключительному шагу.

4. Ищется класс с именем, совпадающим с базовым именем, то есть класс со следующим именем:

   basename

   Он представляет принимаемый по умолчанию набор ресурсов, используемый всеми регионами, которые не поддерживаются явным образом.

На каждом шаге этого процесса метод `getBundle()` ищет сначала файл класса с заданным именем. Если файл класса не найден, он использует метод `getResourceAsStream()` класса `ClassLoader` для поиска файла `Properties` с тем же именем, что и у класса, и расширением *.properties*. Если такой файл свойств найден, его содержимое используется для создания объекта `Properties`, а метод `getBundle()` создает и возвращает экземпляр `PropertyResourceBundle`, экспортирующий свойства файла `Properties` посредством `ResourceBundle` API.

Если метод `getBundle()` не может найти ни класс, ни файл свойств, соответствующие региону, ни на одном из четырех шагов поиска, он по-

вторяет поиск, используя принимаемый по умолчанию регион вместо заданного. Если и на этом шаге соответствующий набор ResourceBundle не найден, getBundle() выдает исключение MissingResourceException.

Для каждого объекта ResourceBundle может быть задан родительский набор ResourceBundle. При поиске заданного ресурса в наборе ResourceBundle метод getObject() сначала ищет в заданном наборе, но если заданный ресурс в этом наборе не определен, он рекурсивно производит поиск в родительском наборе. Таким образом, каждый набор ResourceBundle наследует ресурсы своего родителя и может замещать некоторые из них или все. (Обратите внимание, что мы употребляем термины «наследовать» (inherit) и «замещать» (override) в ином смысле, чем когда говорим о классах, наследующих и замещающих методы своего родительского класса.) Это значит, что необязательно в каждом определяемом наборе ResourceBundle определять все ресурсы, требующиеся приложению. Например, можно определить ResourceBundle с сообщениями для франкоязычных пользователей, а затем уже меньший и более специализированный ResourceBundle, замещающий некоторые из этих сообщений для франкоговорящих канадцев.

Ваше приложение не обязано находить и устанавливать родительский объект для используемого им объекта ResourceBundle. Метод getBundle() на самом деле сделает это за вас. Когда getBundle() найдет подходящий файл класса или свойств, как описано выше, он не возвратит найденный объект ResourceBundle немедленно. Вместо этого он пройдет все оставшиеся шаги приведенного выше процесса поиска, отыскивая менее специфичные файлы классов или свойств, от которых ResourceBundle может унаследовать ресурсы. Найдя эти менее специфичные ресурсы, getBundle() назначает их приемлемыми предками первоначального набора ресурсов. Только перепробовав все возможности, он вернет созданный объект ResourceBundle.

Продолжая начатый выше пример, допустим, что наша программа запущена в провинции Квебек.[1] В этом случае метод getBundle(), вероятно, сначала найдет маленький специализированный класс ResourceBundle, содержащий лишь несколько специфичных для Квебека ресурсов. Затем он обратит внимание на более общий ResourceBundle, содержащий сообщения на французском языке, и назначит этот набор родителем оригинального квебекского набора. Наконец, getBundle() будет искать (и, вероятно, найдет) класс, определяющий принимаемый по умолчанию набор ресурсов, скорее всего, на английском языке (предположив, что английский – родной язык программиста). Этот принимаемый по умолчанию набор назначается родителем французского набора (что сделает его дедушкой квебекского). Когда приложение станет искать требуемый ресурс, квебекский набор будет просматриваться в первую очередь. Если в нем ресурс не определен, просматри-

[1] Франкоговорящая провинция в Канаде. – *Примеч. ред.*

вается французский набор. А все не найденные во французском наборе ресурсы будут отыскиваться в наборе, принимаемом по умолчанию.

Пример набора ResourceBundle

Изучение некоторого кода внесет ясность в обсуждение наборов ресурсов. Пример 7.4 – это вспомогательная подпрограмма для создания меню Swing. По заданному списку имен элементов меню она отыскивает в наборе ресурсов надписи и комбинации горячих клавиш для этих элементов меню и создает локализованное меню. Этот пример содержит простую тестовую программу.

На рис. 7.2 показаны меню, созданные этой программой для американского, британского и французского регионов. Эта программа не может, разумеется, работать без локализованных наборов ресурсов, в которых она отыскивает локализованные надписи для элементов меню.

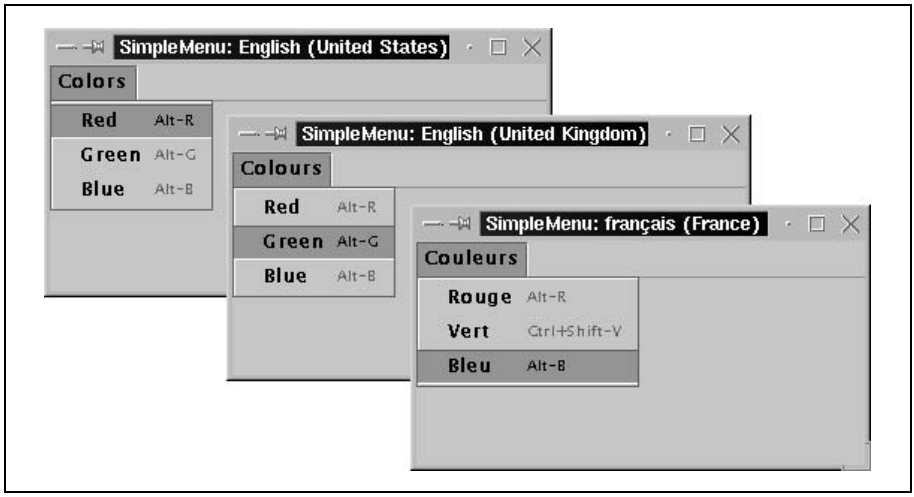


***Рис. 7.2.** Локализованные меню*

Пример 7.4. SimpleMenu.java

```
package com.davidflanagan.examples.i18n;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Locale;
import java.util.ResourceBundle;
import java.util.MissingResourceException;

/** Вспомогательный класс для автоматического создания локализованных меню*/
public class SimpleMenu {
  /** Вспомогательный метод, создающий меню */
  public static JMenu create(ResourceBundle bundle,
```

```
                String menuname, String[] itemnames,
                ActionListener listener)
{
  // Получаем из набора ресурсов заголовок меню.
  // Имя меню назначаем заголовком, принимаемым по умолчанию.
  String menulabel;
  try { menulabel = bundle.getString(menuname + ".label"); }
  catch(MissingResourceException e) { menulabel = menuname; }

  // Создаем меню.
  JMenu menu = new JMenu(menulabel);

  // Для каждого элемента меню.
  for(int i = 0; i < itemnames.length; i++) {
    // Ищем соответствующую надпись, используя имя элемента
    // в качестве принимаемой по умолчанию надписи.
    String itemlabel;
    try {
      itemlabel = bundle.getString(menuname+"."+itemnames[i]+".label");
    }
    catch (MissingResourceException e) { itemlabel = itemnames[i]; }

    JMenuItem item = new JMenuItem(itemlabel);

    // Ищем комбинацию горячих клавиш для элемента меню
    try {
      String acceleratorText =
        bundle.getString(menuname+"."+itemnames[i]+".accelerator");
      item.setAccelerator(KeyStroke.getKeyStroke(acceleratorText));
    }
    catch (MissingResourceException e) {}

    // Регистрируем для элемента слушатель и команду.
    if (listener != null) {
      item.addActionListener(listener);
      item.setActionCommand(itemnames[i]);
    }

    // Вставляем элемент в меню.
    menu.add(item);
  }

  // Возвращаем автоматически созданное локализованное меню.
  return menu;
}

/** Простая тестовая программа для приведенного выше кода */
public static void main(String[] args) {
  // Извлекаем регион: принимаемый по умолчанию или заданный
  // в командной строке
  Locale locale;
  if (args.length == 2) locale = new Locale(args[0], args[1]);
  else locale = Locale.getDefault();

  // Получаем набор ресурсов для этого объекта Locale. Если набор не найден,
```

```
        // будет выдано непроверяемое исключение MissingResourceException.
        ResourceBundle bundle =
          ResourceBundle.getBundle("com.davidflanagan.examples.i18n.Menus",
                        locale);

        // Создаем простое окно GUI для отображения меню
        final JFrame f = new JFrame("SimpleMenu: " +  // Window title
                    locale.getDisplayName(Locale.getDefault()));
        JMenuBar menubar = new JMenuBar();      // Создаем полосу меню.
        f.setJMenuBar(menubar);           // Добавляем полосу меню в окно

        // Определяем слушатель, который будет использоваться нашим меню.
        ActionListener listener = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
              String s = e.getActionCommand();
              Component c = f.getContentPane();
              if (s.equals("red")) c.setBackground(Color.red);
              else if (s.equals("green")) c.setBackground(Color.green);
              else if (s.equals("blue")) c.setBackground(Color.blue);
            }
          };

        // Теперь создаем меню, применяя нашу вспомогательную подпрограмму
        // с созданными нами набором ресурсов и слушателем
        JMenu menu = SimpleMenu.create(bundle, "colors",
                        new String[] {"red", "green", "blue"},
                        listener);

        // Finally add the menu to the GUI, and pop it up
        menubar.add(menu) ;       // Добавляем меню к полосе меню
        f.setSize(300, 150);      // Устанавливаем размер окна.
        f.setVisible(true);       // Отображаем окно.
      }
    }
```

Как я уже говорил, этот пример не будет работать отдельно. Для локализации меню он пользуется набором ресурсов. Приведенный ниже листинг показывает три файла свойств, служащие набором ресурсов для этого примера. Обратите внимание на то, что в этом листинге содержатся тела трех отдельных файлов:

```
# Файл Menus.properties является принимаемым по умолчанию набором
# ресурсов «Menus». Как американский программист, я назначил мой
# регион принимаемым по умолчанию.
colors.label=Colors
colors.red.label=Red
colors.red.accelerator=alt R
colors.green.label=Green
colors.green.accelerator=alt G
colors.blue.label=Blue
colors.blue.accelerator=alt B

# Это файл Menus_en_GB.properties. Он представляет набор ресурсов для
```

```
# британского английского. Обратите внимание на то, что он замещает только
# одно определение ресурса, а остальные наследует от принятого
# по умолчанию (американского) набора.
colors.label=Colours

# Это файл Menus_fr.properties. Это набор ресурсов для всех
# франкоязычных регионов. Он замещает большинство, но не все ресурсы
# в принимаемом по умолчанию наборе.
colors.label=Couleurs
colors.red.label=Rouge
colors.green.label=Vert
colors.green.accelerator=control shift V
colors.blue.label=Bleu
```

Форматированные сообщения

Мы видели, что в целях интернационализации программ все выводимые для пользователя сообщения следует помещать в наборы ресурсов. Это просто, когда подлежащий локализации текст состоит из простых надписей вроде элементов меню и надписей на кнопках. Однако это сложнее, когда сообщения составляются из статического текста и динамических значений. Компилятору, например, может понадобиться вывести сообщение «Ошибка в строке 5 файла hello.java», в котором номер строки и имя файла являются динамическими и не зависят от региона, тогда как остальная часть сообщения является статической и нуждается в локализации.

Класс `MessageFormat` пакета `java.text` потрясающе управляется с такого рода сообщениями. Чтобы им воспользоваться, следует сохранять в `ResourceBundle` только статическую часть сообщения, включив в него специальные символы, помечающие места для вставки динамических частей сообщения. Так, например, один набор ресурсов может содержать сообщение «Ошибка в строке {0} файла {1}». А другой набор ресурсов может содержать «перевод», например «Erreur: {1}: {0}».

Для применения форматированных сообщений такого рода сначала по статической части сообщения создается объект `MessageFormat`, а затем вызывается его метод `format()`, которому передается массив подставляемых значений. В нашем случае массив содержит объект типа `Integer`, задающий номер строки, и объект типа `String`, задающий имя файла. Класс `MessageFormat` знает о других классах `Format`, определенных в `java.text`. Он создает и использует объекты `NumberFormat` для форматирования чисел и объекты `DateFormat` для форматирования дат и времени дня. К тому же можно сконструировать сообщения, создающие объекты `ChoiceFormat` для преобразования чисел в строки. Это бывает полезно при работе с перечислимыми типами (enumerated types), такими как числа, соответствующие названиям месяцев, или при необходимости использовать единственное или множественное число части речи в зависимости от значения некоторого числа.

Пример 7.5 демонстрирует такое использование `MessageFormat`. Это вспомогательный класс с единственным статическим методом для локализованного отображения исключений и сообщений об ошибках. Будучи вызван, этот код пытается загрузить `ResourceBundle` с базовым именем «Errors». Если таковой найден, программа ищет содержащий сообщения ресурс, используя имя класса переданного ему объекта исключения. Если и этот ресурс найден, программа отображает сообщение об ошибке. Методу `format()` передается массив, содержащий пять значений. Локализованное сообщение об ошибке может содержать некоторые или все из этих аргументов.

Определенный в этом примере метод `LocalizedError.display()` использовался в примере 7.2 в начале этой главы. Принимаемый по умолчанию набор ресурсов *Errors.properties*, используемый в этом примере, приведен после листинга программы. Отображение сообщений об ошибках для этой программы прекрасно интернационализировано. Перенос сообщений об ошибках для новых регионов состоит всего лишь в переводе (локализации) файла *Errors.properties*.

Пример 7.5. LocalizedError.java

```
package com.davidflanagan.examples.i18n;
import java.text.*;
import java.io.*;
import java.util.*;

/**
 * Вспомогательный класс, способный выводить локализованное сообщение
 * об исключении, зависящее от класса исключения. Он использует MessageFormat
 * и передает пять аргументов, которые может включать
 * в себя локализованное сообщение:
 *  {0}: сообщение, включенное в исключение или ошибку
 *  {1}: полное имя класса исключения или ошибки
 *  {2}: файл, предположительно являющийся источником ошибки
 *  {3}: номер строки в этом файле
 *  {4}: текущие дата и время дня
 * Сообщения ищутся в наборе ResourceBundle с базовым именем «Errors»,
 * а в качестве имени ресурса используется полное имя класса объекта
 * исключения. Если для данного класса исключений не найдено никакого
 * ресурса, проверяется его родительский класс.
 **/
public class LocalizedError {
  public static void display(Throwable error) {
    ResourceBundle bundle;
    // Пытаемся получить набор ресурсов.
    // При неудаче отображаем сообщение об ошибке в нелокализованном виде.
    try { bundle = ResourceBundle.getBundle("Errors"); }
    catch (MissingResourceException e) {
      error.printStackTrace(System.err);
      return;
    }
```

```
  // В этом наборе ищем ресурс с локализованным сообщением, используя
  // имя класса ошибки (или его родительского класса) в качестве имени
  // ресурса. Если ресурс не найден, отображаем нелокализованное
  // сообщение об ошибке.
  String message = null;
  Class c = error.getClass();
  while((message == null) && (c != Object.class)) {
    try { message = bundle.getString(c.getName()); }
    catch (MissingResourceException e) { c = c.getSuperclass(); }
  }
  if (message == null) { error.printStackTrace(System.err); return; }

  // Пытаемся определить имя файла и номер строки источника исключения.
  // Выводим распечатку стека ошибок в строку, используя эвристическое
  // правило, что первый номер строки, появляющийся в распечатке стека,
  // идет после первого или второго двоеточия. Мы предполагаем, что этот кадр
  // стека – первый, который программист может контролировать, и сообщаем
  // о нем как о местоположении исключения. Обратите внимание на то, что
  // результат здесь зависит от реализации и не является надежным...
  String filename = "";
  int linenum = 0;
  try {
    StringWriter sw = new StringWriter(); // Строковый поток вывода.
    PrintWriter out=new PrintWriter(sw); // PrintWriter на его выходе.
    error.printStackTrace(out);         // Печатаем стек ошибок.
    String trace = sw.toString();       // Преобразуем его в строку.
    int pos = trace.indexOf(':');       // Ищем первое двоеточие.
   if (error.getMessage() != null)    // Если для ошибки имеется сообщение,
      pos = trace.indexOf(':', pos+1); // ищем второе двоеточие.
    int pos2 = trace.indexOf(')', pos);  // Ищем конец номера строки
    linenum = Integer.parseInt(trace.substring(pos+1,pos2)); // номер строки
    pos2 = trace.lastIndexOf('(', pos);  // Назад, к началу имени файла.
    filename = trace.substring(pos2+1, pos); // Получаем имя файла.
  }
  catch (Exception e) { ; }         // Игнорируем исключения.

  // Устанавливаем массив аргументов, используемый вместе с сообщением
  String errmsg = error.getMessage();
  Object[] args = {
    ((errmsg!= null)?errmsg:""), error.getClass().getName(),
    filename, new Integer(linenum), new Date()
  };
  // Наконец, отображаем локализованное сообщение об ошибке, используя
  // MessageFormat.format() для вставки аргументов в сообщение.
  System.out.println(MessageFormat.format(message, args));
}

/**
 * Это простая тестовая программа, демонстрирующая работу метода display().
 * Ее можно использовать для генерирования и отображения
 * FileNotFoundException или ArrayIndexOutOfBoundsException
 **/
```

```
  public static void main(String[] args) {
    try { FileReader in = new FileReader(args[0]); }
    catch(Exception e) { LocalizedError.display(e); }
  }
}
```

Ниже представлен файл свойств набора ресурсов, используемый для локализации сообщений об ошибках, которые могут выдаваться классом ConvertEncoding из начала этой главы:

```
#
# Это файл Errors.properties
# Одно свойство для каждого класса исключений, о котором наша программа
# может сообщить. Обратите внимание на обратный слэш, используемый
# для переноса длинных строк. Обратите внимание также на применение
# символов \n и \t для перехода на новую строку и табуляции
#
java.io.FileNotFoundException: \
Error: File "{0}" not found\n\t\
Error occurred at line {3} of file "{2}"\n\tat {4}

java.io.UnsupportedEncodingException: \
Error: Specified encoding not supported\n\t\
Error occurred at line {3} of file "{2}"\n\tat {4,time} on {4,date}

java.io.CharConversionException:\
Error: Character conversion failure. Input data is not in specified format.

# Общий ресурс. Отображается сообщение для всех ошибок и исключений,
# не обрабатываемых более специфичными ресурсами.
java.lang.Throwable:\
Error: {1}: {0}\n\t\
Error occurred at line {3} of file "{2}"\n\t{4,time,long} {4,date,long}
```

При наличии такого набора ресурсов ConvertEncoding будет выдавать сообщения об ошибках следующего вида:

```
Error: File "myfile (No such file or directory)" not found
      Error occurred at line 64 of file "FileInputStream.java" at 7/9/00 9:28 PM
```

Или для текущего региона fr_FR:

```
Error: File "myfile (Aucun fichier ou repertoire de ce type)" not found
Error occurred at line 64 of file "FileInputStream.java" at 09/07/00 21:28
```

Упражнения

7-1. У некоторых связанных с интернационализацией классов, таких как NumberFormat и DateFormat, есть статические методы с именем getAvailableLocales(), которые возвращают массив поддерживаемых ими объектов Locale. Название страны для данного объекта Locale можно найти при помощи метода getDisplayCountry(). Обра-

тите внимание, что этот метод имеет два варианта. Один вызывается без аргументов и отображает название страны, как оно пишется в регионе, используемом по умолчанию. Другая версия `getDisplayCountry()` ожидает объект `Locale` в качестве аргумента и отображает название страны на языке указанного региона.

Напишите программу, отображающую названия стран для всех регионов, возвращаемых методом `NumberFormat.getAvailableLocales()`. Используя соответствующие различным регионам статические константы, определяемые классом `Locale`, отобразите название каждой страны по-английски, по-французски, по-немецки и по-итальянски.

7-2. Измените класс `Portfolio` из примера 7.3, убрав из кодов все отображаемые строки. Вместо них используйте классы `ResourceBundle` и `MessageFormat`, как это сделано в примерах 7.4 и 7.5.

7-3. Напишите программу цифровых часов, отображающих текущие дату и время дня в Вашингтоне, Лондоне, Париже, Бонне, Пекине и Токио. Отобразите дату и время дня в традиционных для этих городов форматах. Вам понадобится прочитать о классе `java.util.TimeZone` и методе `DateFormat.setTimeZone()`. Справьтесь по карте или загляните в Интернет, чтобы узнать часовые пояса для этих городов. Напишите программу в виде приложения AWT или Swing или в виде апплета, когда прочтете главы 10 и 15. Возможно, вы решите написать эту программу на основе апплета `Clock` из примера 15.2.

7-4. Пример 7.5 показывает, как можно использовать `ResourceBundle` для интернационализации текста в меню ваших приложений. У Swing есть свойство, препятствующее интернационализации – конструкторам `JButton`, `JMenu` и `JMenuItem` наряду с прочими аргументами передаются отображаемые надписи. Для программистов это сильный соблазн включить эти надписи в код программы. Создайте интернационализированные подклассы этих компонентов (назвав их `IButton`, `IMenu` и `ImenuItem`), принимающие в качестве аргументов своих конструкторов имена ресурсов. Каждый класс должен отыскивать набор ресурсов с именем «Labels» и в нем искать надпись для кнопки или меню, соответствующую переданному конструктору имени ресурса. Если набора нет или данный ресурс в нем отсутствует, классы `IButton`, `IMenu` и `IMenuItem` должны в качестве надписей использовать имена ресурсов. Напишите простую тестовую программу (и несколько файлов свойств), демонстрирующую работу этих классов для двух или трех регионов. Перед выполнением этого упражнения вам, наверное, понадобится прочесть главу 10.

Глава 8

Отражение

API Reflection (Отражение) позволяет Java-программам инспектировать самое себя и манипулировать собой; он включает в себя класс `java.lang.Class` и пакет `java.lang.reflect`, представляющие члены некоторого класса при помощи объектов `Method`, `Constructor` и `Field`.

Reflection может получать информацию о классе и его членах. Это технология, которой пользуется, например, механизм интроспекции JavaBeans (см. главу 14 «JavaBeans») для определения свойств, событий и методов, поддерживаемых компонентом. Reflection в Java может также манипулировать объектами. Класс `Field` можно использовать как для опроса содержимого полей объекта, так и для установки их значений, класс `Method` – для вызова методов, а класс `Constructor` – для создания новых объектов. Примеры в этой главе демонстрируют как инспектирование объектов, так и манипулирование ими при помощи API Reflection.

Помимо примеров этой главы API Reflection используется также в примере из главы 17 «Доступ к базам данных при помощи SQL».

Получение информации о классах и членах

Пример 8.1 показывает программу, использующую классы `Class`, `Constructor`, `Field` и `Method` для отображения информации о заданном классе. Вывод этой программы подобен кратким описаниям классов, приведенным в книге «Java in a Nutshell». (Вы можете заметить, что имена аргументов методов не показаны; имена аргументов не сохраняются в файле класса, поэтому они не могут быть получены через API Reflection.)

Здесь представлен вывод, полученный в результате применения ShowClass к нему самому:

```
public class ShowClass extends Object {
 // Constructors
 public ShowClass();
 // Fields
 // Methods
 public static void main(String[]) throws ClassNotFoundException;
 public static String modifiers(int);
 public static void print_class(Class);
 public static String typename(Class);
 public static void print_field(Field);
 public static void print_method_or_constructor(Member);
}
```

Код этого примера совершенно прозрачен. Здесь используется метод Class.forName() для динамической загрузки класса, заданного по имени, а затем вызываются различные методы объекта Class для просмотра родительского класса, интерфейсов и членов класса. В примере используются объекты Constructor, Field и Method для получения информации о каждом члене класса.

Пример 8.1. ShowClass.java

```
package com.davidflanagan.examples.reflect;
import java.lang.reflect.*;

/** Программа, выдающая краткое описание класса, заданного по имени */
public class ShowClass {
  /** Метод main(). Печатает информацию о заданном классе */
  public static void main(String[] args) throws ClassNotFoundException {
    Class c = Class.forName(args[0]);
    print_class(c);
  }

  /**
   * Отображаем модификаторы, имя, родительский класс и интерфейсы класса
   * или интерфейса. Затем выводим список его конструкторов, полей и методов.
   **/
  public static void print_class(Class c)
  {
    // Печатаем модификаторы, тип (класс или интерфейс), имя,
    // родительский класс.
    if (c.isInterface()) {
      // Модификаторы будут здесь включать в себя ключевое слово «interface»...
      System.out.print(Modifier.toString(c.getModifiers()) + " " +
                       typename(c));
    }
    else if (c.getSuperclass() != null) {
      System.out.print(Modifier.toString(c.getModifiers()) + " class " +
                       typename(c) +
```

```
            " extends " + typename(c.getSuperclass()));
    }
    else {
      System.out.print(Modifier.toString(c.getModifiers()) + " class " +
            typename(c));
    }

    // Печатаем интерфейсы или суперинтерфейсы класса или интерфейса.
    Class[] interfaces = c.getInterfaces();
    if ((interfaces != null) && (interfaces.length > 0)) {
      if (c.isInterface()) System.out.print(" extends ");
      else System.out.print(" implements ");
      for(int i = 0; i < interfaces.length; i++) {
        if (i > 0) System.out.print(", ");
        System.out.print(typename(interfaces[i]));
      }
    }

    System.out.println(" {");      // Начало списка членов класса.

    // Теперь просматриваем и печатаем члены класса.
    System.out.println(" // Constructors");
    Constructor[] constructors = c.getDeclaredConstructors();
    for(int i = 0; i < constructors.length; i++)  // Выводим конструкторы.
      print_method_or_constructor(constructors[i]);

    System.out.println(" // Fields");
    Field[] fields = c.getDeclaredFields();       // Просматриваем поля.
    for(int i = 0; i < fields.length; i++)        // Отображаем их.
      print_field(fields[i]);

    System.out.println(" // Methods");
    Method[] methods = c.getDeclaredMethods();     // Просматриваем методы.
    for(int i = 0; i < methods.length; i++)        // Отображаем их.
      print_method_or_constructor(methods[i]);

    System.out.println("}");       // Конец списка членов класса.
  }

  /** Возвращаем имя интерфейса или примитивного типа, обрабатывая массивы. */
  public static String typename(Class t) {
    String brackets = "";
    while(t.isArray()) {
      brackets += "[]";
      t = t.getComponentType();
    }
    String name = t.getName();
    int pos = name.lastIndexOf('.');
    if (pos != -1) name = name.substring(pos+1);
    return name + brackets;
  }

  /** Возвращаем строковую версию модификаторов,
      для красоты применяя пробелы. */
```

```
  public static String modifiers(int m) {
    if (m == 0) return "";
    else return Modifier.toString(m) + " ";
  }

  /** Печатаем модификаторы, тип и имя поля. */
  public static void print_field(Field f) {
    System.out.println(" " + modifiers(f.getModifiers()) +
               typename(f.getType()) + " " + f.getName() + ";");
  }

  /**
   * Печатаем модификаторы, возвращаем тип, имя, типы параметров и типы
   * исключений метода или конструктора. Обратите внимание на применение
   * интерфейса Member, позволяющее этому методу работать как
   * с объектами Method, так и с объектами Constructor
   **/
  public static void print_method_or_constructor(Member member) {
    Class returntype=null, parameters[], exceptions[];
    if (member instanceof Method) {
      Method m = (Method) member;
      returntype = m.getReturnType();
      parameters = m.getParameterTypes();
      exceptions = m.getExceptionTypes();
      System.out.print(" " + modifiers(member.getModifiers()) +
                typename(returntype) + " " + member.getName() +
                "(");
    } else {
      Constructor c = (Constructor) member;
      parameters = c.getParameterTypes();
      exceptions = c.getExceptionTypes();
      System.out.print(" " + modifiers(member.getModifiers()) +
                typename(c.getDeclaringClass()) + "(");
    }

    for(int i = 0; i < parameters.length; i++) {
      if (i > 0) System.out.print(", ");
      System.out.print(typename(parameters[i]));
    }
    System.out.print(")");
    if (exceptions.length > 0) System.out.print(" throws ");
    for(int i = 0; i < exceptions.length; i++) {
      if (i > 0) System.out.print(", ");
      System.out.print(typename(exceptions[i]));
    }
    System.out.println(";");
  }
}
```

Вызов метода, заданного по имени

Пример 8.2 определяет класс Command, демонстрирующий еще одно использование Reflection API. Объект Command включает в себя объект Method, объект, метод которого вызывается, и массив переменных, передаваемых этому методу. Метод invoke() вызывает метод заданного объекта, используя заданные аргументы. Метод actionPerformed() делает то же самое. Если вы уже прочитали главу 10 «Графические интерфейсы пользователя (GUI)», вы знаете, что этот метод реализует интерфейс java.awt.event.ActionListener, из чего следует, что объекты Command могут использоваться как слушатели действий, позволяя реагировать на нажатие кнопок, выбор пунктов меню и другие события, относящиеся к графическому интерфейсу пользователя. Для того чтобы обработать события, GUI-программы обычно создают набор реализаций интерфейса ActionListener. При использовании класса Command слушатели действий могут определяться без необходимости создания множества новых классов.

Самой полезной вещью (при этом с самым сложным кодом) в классе Command является метод parse(), разбирающий строку, которая содержит имя метода и список аргументов, для создания объекта Command. Он полезен, например, потому что позволяет считывать объекты Command из конфигурационных файлов. Мы воспользуемся этой возможностью класса Command в главе 10.

Java не позволяет передавать методы прямо, подобно данным, но Reflection API делает возможным косвенный вызов методов, заданных их именами. Заметим, что этот прием не особенно эффективен. Но для асинхронной обработки событий в GUI он достаточно эффективен: косвенные вызовы методов через Reflection API действуют гораздо быстрее, чем требуется для удовлетворения ограниченных потребностей человеческого восприятия. Однако вызов методов по имени оказывается неприемлемым, когда необходимы повторные вызовы или когда компьютер не ожидает ввода от человека. Таким образом, этот прием не следует использовать, например, для передачи метода сравнения в процедуру сортировки или для передачи фильтра имен файлов методу, выдающему листинг каталога. В подобных случаях следует применять стандартную технику реализации класса, содержащего нужный метод, и передачи экземпляра класса соответствующей процедуре.

Пример 8.2. Command.java

```
package com.davidflanagan.examples.reflect;
import java.awt.event.*;
import java.beans.*;
import java.lang.reflect.*;
import java.io.*;
import java.util.*;
```

```
/**
 * Этот класс представляет объект Method, список аргументов, передаваемых
 * этому методу, и объект, из которого этот метод должен вызываться.
 * Метод invoke() вызывает метод. Метод actionPerformed() делает то же
 * самое, позволяя этому классу реализовывать ActionListener и использоваться
 * для реакции на события ActionEvents, генерируемые в GUI или где-либо еще.
 * Статический метод parse() разбирает строку, представляющую метод,
 * и его аргументы.
 **/
public class Command implements ActionListener {
  Method m;      // Вызываемый метод
  Object target; // Объект, из которого вызывается метод
  Object[] args; // Аргументы, передаваемые методу

  // Пустой массив; используется для методов, не имеющих аргументов.
  static final Object[] nullargs = new Object[] {};

  /** Этот конструктор создает объект Command для метода без аргументов */
  public Command(Object target, Method m) { this(target, m, nullargs); }

  /**
   * Этот конструктор создает объект Command для метода, принимающего
   * заданный массив аргументов. Отметим, что метод parse() предоставляет
   * еще один способ создания объекта Command.
   **/
  public Command(Object target, Method m, Object[] args) {
    this.target = target;
    this.m = m;
    this.args = args;
  }

  /**
   * Вызывает Command посредством вызова метода из его объекта target
   * и передачи аргументов. См. также actionPerformed(), который,
   * в отличие от этого метода, не выдает проверяемых исключений.
   **/
  public void invoke()
    throws IllegalAccessException, InvocationTargetException
  {
    m.invoke(target, args); // Используем отражение для вызова метода
  }

  /**
   * Этот метод реализует интерфейс ActionListener. Он подобен методу invoke()
   * с тем отличием, что перехватывает выданные этим методом исключения
   * и передает их в виде непроверяемого исключения RuntimeException
   **/
  public void actionPerformed(ActionEvent e) {
    try {
      invoke();              // Обращаемся к вызывающему методу,
    }
    catch (InvocationTargetException ex) { // но обрабатываем исключения
      throw new RuntimeException("Command: " +
```

```
                    ex.getTargetException().toString());
    }
    catch (IllegalAccessException ex) {
      throw new RuntimeException("Command: " + ex.toString());
    }
  }

  /**
   * Этот статический метод создает объект Command с использованием заданного
   * объекта target и заданной строки. Эта строка должна содержать имя
   * метода, за которым следуют необязательный, заключенный в скобки список
   * аргументов, разделенных запятыми, и точка с запятой. Аргументы могут
   * быть литералами логическими, целыми или типа double, или заключенными
   * в двойные кавычки строками. Анализатор терпимо относится к пропущенным
   * запятым, точкам с запятой и кавычкам, но выдает IOException,
   * если ему не удается разобрать строку.
   **/
  public static Command parse(Object target, String text) throws IOException
  {
    String methodname;                  // Имя метода
    ArrayList args = new ArrayList();   // Будет содержать
                                        // разобранные аргументы.
    ArrayList types = new ArrayList(); // Будет содержать типы аргументов.

    // Преобразуем строку в символьный поток и применяем класс
    // StreamTokenizer для преобразование его в поток лексем
    // (маркеров, tokens)
    StreamTokenizer t = new StreamTokenizer(new StringReader(text));

    // Первой лексемой должно быть имя метода
    int c = t.nextToken();    // читаем лексему
    if (c != t.TT_WORD)       // проверяем ее тип
      throw new IOException("Отсутствует имя метода для команды");
    methodname = t.sval;      // Запоминаем имя метода

    // Далее должна следовать либо точка с запятой, либо открывающая скобка
    c = t.nextToken();
    if (c == '(') { // Увидев открывающую скобку, разбираем список аргументов
      for(;;) {               // Цикл до конца списка аргументов
        c = t.nextToken();    // Читаем следующую лексему

        if (c == ')') {       // Проверяем, не конец ли это списка.
          c = t.nextToken(); // Если да, выделяем необязательную
                             // точку с запятой
          if (c != ';') t.pushBack();
          break;              // Выходим из цикла.
        }

        // Если нет, лексема является аргументом; определяем ее тип
        if (c == t.TT_WORD) {
          // Если лексема является идентификатором (identifier), выделяем
          // логические литералы, а все другие лексемы интерпретируем
          // как строковые литералы без кавычек.
          if (t.sval.equals("true")) {    // Логический литерал
```

```
          args.add(Boolean.TRUE);
          types.add(boolean.class);
        }
        else if (t.sval.equals("false")) { // Логический литерал
          args.add(Boolean.FALSE);
          types.add(boolean.class);
        }
        else {               // Считаем, что это строка
          args.add(t.sval);
          types.add(String.class);
        }
      }
      else if (c == '"') {     // Если лексема - это строка в кавычках
        args.add(t.sval);
        types.add(String.class);
      }
      else if (c == t.TT_NUMBER) { // Если лексема - это число
        int i = (int) t.nval;
        if (i == t.nval) {      // Проверяем, целое ли оно
          // Замечание: этот код проинтерпретирует лексему
          // вида "2.0" как целое число!
          args.add(new Integer(i));
          types.add(int.class);
        }
        else {      // В противном случае оно принадлежит типу double
          args.add(new Double(t.nval));
          types.add(double.class);
        }
      }
      else {            // Любая другая лексема является ошибкой
        throw new IOException("Неизвестная лексема " + t.sval +
                    " в списке аргументов метода " +
                    methodname + "().");
      }

      // Далее должна следовать запятая, но если это окажется не так,
      // мы жаловаться не будем
      c = t.nextToken();
      if (c != ',') t.pushBack();
    }
  }
  else if (c != ';') { // Если за именем метода не следует запятая,
    t.pushBack();      // ожидаем точку с запятой, но не настаиваем на этом.
  }

  // Мы разобрали список аргументов.
  // Теперь преобразуем списки аргументов и их типов в массивы
  Object[] argValues = args.toArray();
  Class[] argtypes = (Class[])types.toArray(new Class[argValues.length]);

  // В этом месте у нас уже есть имя метода и массивы типов и значений
  // аргументов. Применяем отражение к классу объекта target,
```

```
    // чтобы найти метод с заданными именем и типами аргументов. Если
    // метода с заданным именем не нашлось, выдаем исключение.
    Method method;
    try { method = target.getClass().getMethod(methodname, argtypes); }
    catch (Exception e) {
      throw new IOException("Нет такого метода или неправильно заданы " +
                  "типы аргументов: " + methodname);
    }

    // Наконец, создаем и возвращаем объект Command, используя переданный
    // этому методу объект target, полученный выше объект Method и
    // массив аргументов, выделенный из строки.
    return new Command(target, method, argValues);
  }

  /**
   * Эта простая программа показывает, как объект Command может быть выделен
   * из строки и использован в качестве объекта ActionListener
   * в Swing-приложении.
   **/
  static class Test {
    public static void main(String[] args) throws IOException {
      javax.swing.JFrame f = new javax.swing.JFrame("Command Test");
      javax.swing.JButton b1 = new javax.swing.JButton("Tick");
      javax.swing.JButton b2 = new javax.swing.JButton("Tock");
      javax.swing.JLabel label = new javax.swing.JLabel("Hello world");
      java.awt.Container pane = f.getContentPane();

      pane.add(b1, java.awt.BorderLayout.WEST);
      pane.add(b2, java.awt.BorderLayout.EAST);
      pane.add(label, java.awt.BorderLayout.NORTH);

      b1.addActionListener(Command.parse(label, "setText(\"tick\");"));
      b2.addActionListener(Command.parse(label, "setText(\"tock\");"));

      f.pack();
      f.show();
    }
  }
}
```

Упражнения

8-1. Напишите программу, получающую в качестве заданного в командной строке аргумента класс Java и использующую класс `Class` для печати всех родительских классов этого класса. Вызванная, например, с аргументом «java.awt.Applet», эта программа должна напечатать следующее: `java.lang.Object java.awt.Component java.awt.Container java.awt.Panel`.

8-2. Переделайте программу, написанную вами в упражнении 8-1, чтобы она распечатывала все интерфейсы, реализованные задан-

ным классом или любым из его родительских классов. Обработайте случай, когда классы реализуют интерфейсы, дочерние по отношению к другим интерфейсам. Если класс реализует, например, `java.awt.LayoutManager2`, а интерфейсы `LayoutManager2` и `LayoutManager` являются родительскими по отношению к первому, они тоже должны быть включены в список.

8-3. На основе класса `Command` из примера 8.2 определите класс `Assignment`. Вместо вызова заданного по имени метода, как это делает `Command`, `Assignment` должен присваивать значение заданному по имени полю объекта. Пусть у вашего класса будет конструктор со следующей сигнатурой:

```
public Assignment(Object target, Field field, Object value)
```

Включите в него также метод `assign()`, который, будучи вызван, присвоит заданное значение заданному полю заданного объекта. Класс `Assignment` должен реализовать `ActionListener` и определять метод `actionPerformed()`, также выполняющий присваивание. Напишите для `Assignment` статический метод `parse()`, подобный методу `parse()` для `Command`. Он должен разбирать строки вида *`fieldname=value`* и уметь выделять логические, числовые и строковые значения.

Глава 9

Сериализация объектов

Сериализация объектов состоит в способности класса `Serializable` выводить состояние экземпляра объекта в байтовый поток, а позже снова считывать это состояние, создавая копию исходного объекта. При сериализации объекта вместе с ним подвергается сериализации вся иерархия объектов, на которые ссылается сериализуемый объект. Это обеспечивает возможность сериализации сложных структур данных, таких как двоичные деревья. Также можно сериализовать апплеты и полные иерархии компонентов GUI.

Простая сериализация

Несмотря на мощь и важность сериализации, она осуществляется с использованием простого API, входящего в пакет `java.io`: объект сериализуется при помощи метода `writeObject()` класса `ObjectOutputStream` и десериализуется методом `readObject()` класса `ObjectInputStream`. Эти классы являются байтовыми потоками, подобными различным другим потокам, какие мы видели в главе 3 «Ввод/вывод». Они реализуют интерфейсы `ObjectOutput` и `ObjectInput` соответственно, а эти интерфейсы являются в свою очередь дочерними для интерфейсов `DataOutput` и `DataInput`. Это значит, что `ObjectOutputStream` определяет те же методы, что и `DataOutputStream` для записи примитивных значений, тогда как `ObjectInputStream` определяет те же методы, что и `DataInputStream`, для чтения примитивных значений. Однако здесь нас интересуют методы `writeObject()` и `readObject()`, которые записывают и считывают объекты.

Сериализовать можно только объекты, реализующие интерфейс `java.io.Serializable`. Интерфейс `Serializable` — это пустой интерфейс; он не определяет никаких методов, подлежащих реализации. Тем не менее

для некоторых классов раскрытие их состояния при использовании механизма сериализации оказывается нежелательным по соображениям безопасности. Поэтому, чтобы класс реализовал этот интерфейс, он должен явно объявить себя в качестве сериализуемого.

Объект сериализуется путем передачи его методу `writeObject()` класса `ObjectOutputStream`. Он выводит значения всех полей объекта, включая закрытые (private) поля и поля, унаследованные от родительских классов. Значения примитивных полей просто записываются в поток, как это делалось бы с `DataOutputStream`. Однако когда поле объекта ссылается на другой объект, массив или строку, метод `writeObject()` рекурсивно вызывается для сериализации также и этого объекта. Если этот объект (или элемент массива) ссылается еще на один объект, `writeObject()` снова рекурсивно вызывается. Таким образом, один вызов `writeObject()` может в результате привести к сериализации всей иерархии объектов. Когда два или большее число объектов ссылаются друг на друга, алгоритм сериализации обеспечивает лишь по одному выводу для каждого объекта; `writeObject()` не может войти в бесконечную рекурсию.

Десериализация объекта просто проходит этот процесс в обратном направлении. Объект считывается из потока данных посредством вызова метода `readObject()` класса `ObjectInputStream`. Он восстанавливает объект в том состоянии, в котором он находился при сериализации. Если объект ссылается на другие объекты, они также рекурсивно десериализуются.

Пример 9.1 демонстрирует основы сериализации. Этот пример определяет общие методы, способные сохранять в файле и получать из него состояние любого сериализуемого объекта. Также он включает в себя интересный метод `deepclone()`, использующий сериализацию для копирования иерархии объектов. В пример входят внутренний класс `Serializable` и тестовый класс, демонстрирующий эти методы в работе.

Пример 9.1. Serializer.java

```
package com.davidflanagan.examples.serialization;
import java.io.*;

/**
 * Этот класс определяет утилиты, использующие Java-сериализацию.
 **/
public class Serializer {
    /**
     * Сериализуем объект o (и все объекты, на которые он ссылается
     * и объявленные как Serializable) и его состояние сохраняем в файле f.
     **/
    static void store(Serializable o, File f) throws IOException {
        ObjectOutputStream out =  // Класс для сериализации
            new ObjectOutputStream(new FileOutputStream(f));
        out.writeObject(o);       // Этот метод сериализует иерархию объектов
        out.close();
    }
```

```
/**
 * Десериализируем содержимое файла f и возвращаем результирующий объект
 **/
static Object load(File f) throws IOException, ClassNotFoundException {
    ObjectInputStream in =    // Класс для десериализации
        new ObjectInputStream(new FileInputStream(f));
    return in.readObject();   // Этот метод десериализует иерархию объектов
}

/**
 * Используем сериализацию объектов для создания полного клона
 * («deep clone») объекта o. Этот метод сериализует объект o и все
 * объекты,на которые тот ссылается, а затем десериализует эту иерархию
 * объектов, т. е. все копируется. Метод deepclone() отличается от метода
 * clone(), который обычно реализуется так, чтобы создавался частичный
 * клон («shallow» clone), копирующий ссылки на другие объекты, а не сами
 * объекты, на которые направлены ссылки
 **/
static Object deepclone(final Serializable o)
    throws IOException, ClassNotFoundException
{
    // Создаем пару канальных (pipe) потоков.
    // Будем записывать байты в один из них, а затем читать из другого.
    final PipedOutputStream pipeout = new PipedOutputStream();
    PipedInputStream pipein = new PipedInputStream(pipeout);

    // Теперь создаем отдельный поток исполнения для сериализации объекта
    // и записи его байтов в PipedOutputStream
    Thread writer = new Thread() {
            public void run() {
                ObjectOutputStream out = null;
                try {
                    out = new ObjectOutputStream(pipeout);
                    out.writeObject(o);
                }
                catch(IOException e) {}
                finally {
                    try { out.close(); } catch (Exception e) {}
                }
            }
        };
    writer.start();  // Запускаем поток исполнения на сериализацию и запись

    // В то же время в этом потоке исполнения читаем и десериализуем
    // данные из канального потока ввода. Получившийся в результате
    // объект будет полным клоном оригинала.
    ObjectInputStream in = new ObjectInputStream(pipein);
    return in.readObject();
}

/**
 * Это простая сериализуемая структура данных, используемая ниже
 * для тестирования определенных выше методов.
 **/
```

```
    public static class DataStructure implements Serializable {
        String message;
        int[] data;
        DataStructure other;
        public String toString() {
            String s = message;
            for(int i = 0; i < data.length; i++)
                s += " " + data[i];
            if (other != null) s += "\n\t" + other.toString();
            return s;
        }
    }

    /** Этот класс определяет тестирующий метод main()*/
    public static class Test {
        public static void main(String[] args)
            throws IOException, ClassNotFoundException
        {
            // Создаем простую иерархию объектов
            DataStructure ds = new DataStructure();
            ds.message = "hello world";
            ds.data = new int[] { 1, 2, 3, 4 };
            ds.other = new DataStructure();
            ds.other.message = "nested structure";
            ds.other.data = new int[] { 9, 8, 7 };

            // Отображаем исходную иерархию
            System.out.println("Исходная структура данных: " + ds);

            // Выводим ее в файл
            File f = new File("datastructure.ser");
            System.out.println("Сохранение в файле...");
            Serializer.store(ds, f);

            // Считываем ее из файла обратно и снова отображаем
            ds = (DataStructure) Serializer.load(f);
            System.out.println("Считано из файла: " + ds);

            // Создаем полный клон и отображаем его. После создания копии
            // переделываем оригинал, чтобы доказать, что клон «полный».
            DataStructure ds2 = (DataStructure) Serializer.deepclone(ds);
            ds.other.message = null; ds.other.data = null; // Change original
            System.out.println("Полный клон: " + ds2);
        }
    }
}
```

Специальная сериализация

Не всякая часть состояния программы может или должна сериализоваться. Объектам `FileDescriptor`**, например, присуща зависимость от платформы или от виртуальной машины. Будучи сериализованным,** `FileDescriptor` **не имел бы, например, никакого смысла на другой вир-**

туальной машине. По этой причине, а также в силу вышеописанных важных соображений безопасности не все объекты могут быть сериализованы.

Даже когда объект является сериализуемым, полная сериализация его состояния может не иметь смысла. Некоторые поля могут быть «черновыми» – в них могут храниться временные или заранее вычисленные значения, не содержащие информации, действительно нужной при десериализации объекта. Рассмотрим компонент GUI. Он может определять поля, в которых хранятся координаты последнего полученного им щелчка мыши. Эта информация не представляет никакого интереса при десериализации компонента, поэтому нет смысла утруждать себя сохранением значений этих полей как части состояния компонента. Чтобы сообщить механизму сериализации, что поле не нужно сохранять, его просто объявляют как `transient` (временное):

```
protected transient short last_x, last_y; // Временные поля координат
                                          положения указателя мыши
```

Бывает также, что поле не является временным (другими словами, оно сохраняет важную часть состояния объекта), но по некоторым причинам оно не может быть успешно сериализовано. Рассмотрим другой компонент GUI, вычисляющий свой предпочтительный размер в зависимости от размера отображаемого им текста. Поскольку размер шрифтов несколько варьируется от платформы к платформе, заранее вычисленный предпочтительный размер перестает быть действительным, если компонент сериализован на платформе одного типа, а десериализован на платформе другого типа. Поскольку на поля с предпочтительным размером нельзя полагаться при десериализации, их следует объявлять как `transient`, чтобы они не занимали места в сериализованном объекте. Однако в этом случае их значения после десериализации объекта должны быть пересчитаны.

Класс может определять особенности поведения своих объектов при сериализации и десериализации (подобные пересчету предпочтительного размера) путем реализации методов `writeObject()` и `readObject()`. Как ни удивительно, эти методы не определены никаким интерфейсом, и их следует определять как `private`. Если класс определяет эти методы, при сериализации и десериализации объектов `ObjectOutputStream` или `ObjectInputStream` они будут вызывать соответствующий метод.

Компонент GUI мог бы, например, определить метод `readObject()`, чтобы дать ему возможность при десериализации пересчитать предпочтительные размеры. Этот метод мог бы выглядеть так:

```
private void readObject(ObjectInputStream in)
       throws IOException, ClassNotFoundException
{
 in.defaultReadObject();      // Обычная десериализация компонента.
 this.computePreferredSize(); // Но теперь его размеры пересчитываются.
}
```

Этот метод вызывает метод `defaultReadObject()` **класса** `ObjectInputStream` **для обычной десериализации объекта, а затем выполняет всю необходимую постобработку.**

Пример 9.2 – более сложный пример специальной сериализации. Он показывает класс, реализующий динамический массив целых чисел. Этот класс определяет метод `writeObject()` **для выполнения некоторой предварительной обработки перед тем, как он будет сериализован, и метод** `readObject()` **– для дополнительной обработки после десериализации.**

Пример 9.2. IntList.java

```
package com.davidflanagan.examples.serialization;
import java.io.*;

/**
 * Простой класс, реализующий динамический массив целых чисел и умеющий
 * сериализовать себя так же эффективно, как массив фиксированного размера.
 **/
public class IntList implements Serializable {
  protected int[] data = new int[8]; // Массив для хранения чисел.
  protected transient int size = 0;  // Индекс следующего неиспользованного
                                     // элемента массива

  /** Возвращается элемент массива */
  public int get(int index) throws ArrayIndexOutOfBoundsException {
    if (index >= size) throw new ArrayIndexOutOfBoundsException(index);
    else return data[index];
  }

  /** К массиву добавляется новое число; при необходимости размер
      массива увеличивается */
  public void add(int x) {
    if (data.length==size) resize(data.length*2); // При необходимости
                                                  // увеличиваем размер массива.
    data[size++] = x;                             // Сохраняем целое здесь.
  }

  /** Внутренний метод для изменения размера выделенного массиву пространства */
  protected void resize(int newsize) {
    int[] newdata = new int[newsize];           // Создаем новый массив
    System.arraycopy(data, 0, newdata, 0, size); // Копируем элементы массива.
    data = newdata;                             // Заменяем старый массив
  }

  /** Перед сериализацией массива освобождаемся от неиспользуемых
      элементов массива */
  private void writeObject(ObjectOutputStream out) throws IOException {
    if (data.length > size) resize(size); // Сжимаем массив.
    out.defaultWriteObject();       // Затем записываем его обычным образом.
  }

  /** Вычисляем временное поле size после десериализации массива */
  private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException
```

```
    {
      in.defaultReadObject();      // Считываем массив обычным образом.
      size = data.length;          // Восстанавливаем значение временного поля.
    }

    /**
     * Содержит ли объект this те же значения, что и объект o? Мы замещаем
     * этот метод класса Object так, чтобы протестировать наш класс.
     **/
    public boolean equals(Object o) {
      if (!(o instanceof IntList)) return false;
      IntList that = (IntList) o;
      if (this.size != that.size) return false;
      for(int i = 0; i < this.size; i++)
        if (this.data[i] != that.data[i]) return false;
      return true;
    }

    /** Метод main() доказывает, что все это работает */
    public static void main(String[] args) throws Exception {
      IntList list = new IntList();
      for(int i = 0; i < 100; i++) list.add((int)(Math.random()*40000));
      IntList copy = (IntList)Serializer.deepclone(list);
      if (list.equals(copy)) System.out.println("equal copies");
      Serializer.store(list, new File("intlist.ser"));
    }
  }
```

Экстернализируемые классы

Интерфейс `Externalizable` **расширяет интерфейс** `Serializable` **и определяет методы** `writeExternal()` **и** `readExternal()`**. Объект** `Externalizable` **может быть сериализован, как и другие объекты** `Serializable`**, но механизм сериализации вызывает методы** `writeExternal()` **и** `readExternal()` **для выполнения сериализации и десериализации. В отличие от методов** `readObject()` **и** `writeObject()` **из примера 9.2, методы** `readExternal()` **и** `writeExternal()` **не могут вызывать методы** `defaultReadObject()` **и** `defaultWriteObject()` **— они должны считывать и записывать все состояние объекта самостоятельно.**

Объявлять объект как `Externalizable` **оказывается полезным, когда у объекта уже есть свой формат файла или когда нужно выполнить нечто, что просто невозможно сделать посредством стандартных методов сериализации. Пример 9.3 определяет класс** `CompactIntList`**, производный от интерфейса** `Externalizable` **и класса** `IntList` **из предыдущего примера. Предполагается, что класс** `CompactIntList` **будет обычно использоваться для хранения большого количества маленьких целых чисел; он реализует интерфейс** `Externalizable`**, поэтому он может определить значительно более компактную форму сериализации, чем формат, используемый классами** `ObjectOutputStream` **и** `ObjectInputStream`**.**

Пример 9.3. CompactIntList.java

```
package com.davidflanagan.examples.serialization;
import java.io.*;

/**
 * Этот подкласс IntList предполагает, что большая часть чисел, которые он
 * содержит, меньше 32 000. Чтобы использовать преимущества такой ситуации,
 * он реализует интерфейс Externalizable и определяет
 * более компактный формат сериализации.
 **/
public class CompactIntList extends IntList implements Externalizable {
  /**
   * Этот номер версии появляется здесь на тот случай, если последующим
   * версиям этого класса понадобится модифицировать формат
   * экстернализации, но вместе с тем сохранить совместимость
   * с объектами, экстернализированными по этой версии
   **/
  static final byte version = 1;

  /**
   * Этот метод из интерфейса Externalizable отвечает за полное сохранение
   * состояния объекта в заданном потоке (stream). Он может писать
   * как угодно, лишь бы readExternal() смог прочитать написанное.
   **/
  public void writeExternal(ObjectOutput out) throws IOException {
    if (data.length > size) resize(size); // Сжимаем массив.

    out.writeByte(version); // Начинаем с номера нашей версии.
    out.writeInt(size);     // Выводим количество элементов массива
    for(int i = 0; i < size; i++) { // Теперь цикл по элементам массива
      int n = data[i];              // Записываемый элемент массива
      if ((n < Short.MAX_VALUE) && (n > Short.MIN_VALUE+1)) {
        // Если значение n помещается в тип short и не равно Short.MIN_VALUE,
        // записываем его как short, экономя при этом два байта
        out.writeShort(n);
      }
      else {
        // В противном слуае сначала выводим специальное значение
        // Short.MIN_VALUE, сигнализируя этим, что число не поместилось
        // в short, а затем выводим число, используя все 4 байта.
        // Всего получается 6 байтов.
        out.writeShort(Short.MIN_VALUE);
        out.writeInt(n);
      }
    }
  }

  /**
   * Этот метод из интерфейса Externalizable отвечает за полное восстановление
   * состояния объекта. Для воссоздания объекта будет вызываться конструктор
   * без аргументов, а этот метод должен прочитать написанное
   * writeExternal(), чтобы восстановить состояние объекта.
   **/
```

```
  public void readExternal(ObjectInput in)
    throws IOException, ClassNotFoundException
  {
    // Начинаем с чтения и проверки номера версии.
    byte v = in.readByte();
    if (v != version)
      throw new IOException("CompactIntList: неизвестный номер версии");

    // Считываем количество элементов массива и создаем массив
    // соответствующей величины
    int newsize = in.readInt();
    resize(newsize);
    this.size = newsize;

    // Теперь считываем из потока соответствующее количество значений
    for(int i = 0; i < newsize; i++) {
      short n = in.readShort();
      if (n != Short.MIN_VALUE) data[i] = n;
      else data[i] = in.readInt();
    }
  }

  /** Метод main()доказывает, что все это работает */
  public static void main(String[] args) throws Exception {
    CompactIntList list = new CompactIntList();
    for(int i = 0; i < 100; i++) list.add((int)(Math.random()*40000));
    CompactIntList copy = (CompactIntList)Serializer.deepclone(list);
    if (list.equals(copy)) System.out.println("equal copies");
    Serializer.store(list, new File("compactintlist.ser"));
  }
}
```

Сериализация и учет версий класса

Одной из особенностей примера 9.3 является включение номера версии в записываемый им поток (stream) сериализации. Это оказывается полезным, если класс со временем развивается и ему приходится использовать при сериализации новый формат. Номер версии позволяет будущим версиям класса распознавать сериализованные объекты, записанные этой версией класса.

Для объектов `Serializable`, не являющихся объектами `Externalizable`, интерфейс Serialization API сам поддерживает учет версий. Для сериализуемого объекта некоторая информация о классе этого объекта должна, очевидно, сериализоваться вместе с ним, чтобы при десериализации объекта мог быть загружен правильный файл класса. Эта информация о классе представляется классом `java.io.ObjectStreamClass`. Он содержит полное имя класса и номер версии для этого класса. Номер версии очень важен, поскольку более ранние версии класса, возможно, окажутся неспособными десериализовать сериализованный

экземпляр, созданный более поздней версией этого же класса. Номер версии класса имеет тип `long`. По умолчанию механизм сериализации создает уникальный номер версии посредством вычисления хеш-ключа из имени класса, имени его родительского класса и всех интерфейсов, которые он реализует, имен и типов его полей и из имен и типов его незакрытых (nonprivate) методов. Таким образом, как только в классе добавляется новый метод, изменяется имя поля или производится даже малейшее изменение API или реализации класса, его вычисляемая версия также меняется. Когда объект сериализован одной версией класса, он не может быть десериализован его версией с другим номером.

Таким образом, внесение изменений в сериализуемый класс, даже самых незначительных, не влияющих на формат сериализации, разрушает совместимость между версиями по сериализации. Например, наш класс `IntList` в действительности обязан иметь метод `set()`, устанавливающий значение заданного элемента списка. Но если добавить этот метод, новая версия класса не сможет десериализовать объекты, сериализованные старой версией. Предупредить эту проблему можно, задав версию класса явно. Это делается путем включения в класс константного поля `serialVersionUID`.

Значение этого поля не важно; оно только должно быть одним и тем же для всех версий класса, имеющих совместимый формат сериализации. Поскольку исходный класс `IntList` из примера 9.2 не содержит поля `serialVersionUID`, его номер версии был неявно вычислен на основе API этого класса. Чтобы дать новой версии класса `IntList` номер, совпадающий с номером оригинальной версии, используется утилита *serialver*, входящая в состав Java SDK:

```
% serialver com.davidflanagan.examples.serialization.IntList
IntList:  static final long serialVersionUID = 4538804519406678841L;
```

Теперь можно запустить *serialver* и задать номер оригинальной версии класса. *serialver* печатает определение поля, пригодное для включения в модифицированную версию класса. Включив это константное поле в модифицированный класс, вы восстановите совместимость по сериализации между новой и первоначальной версией.

Дополнительные возможности учета версий

Иногда в класс вносятся исправления, изменяющие способ сохранения классом своего состояния. Вообразим класс `Rectangle`, представляющий прямоугольник как координаты верхнего левого угла плюс его ширина и высота. Предположим теперь, что класс был заново реализован так, что он сохраняет прежний открытый (public) интерфейс, но теперь прямоугольник представляется двумя точками: координатами верхнего левого и правого нижнего углов. Внутренние закрытые поля этого класса изменились, так что должно казаться, что совместимость

по сериализации между двумя реализациями этого класса попросту невозможна.

В Java 1.2 и более поздних версий, однако, механизм сериализации был усовершенствован, что позволило разделить формат сериализации и поля, используемые в конкретной реализации класса. Класс может теперь объявить закрытое поле `serialPersistentFields`, ссылающееся на массив объектов `java.io.ObjectStreamField`. Каждый из этих объектов определяет имя поля и тип поля. Эти поля не должны быть как-то связаны с полями, реализованными в классе; они являются полями сериализованной формы класса. Путем определения этого массива объектов `ObjectStreamField` класс задает свой формат сериализации. При определении новой версии класса эта новая версия должна быть способна сохранять и восстанавливать свое состояние в формате, определяемом массивом `serialPersistentFields`.

Приемы чтения и записи полей сериализации, объявленных массивом `serialPersistentFields`, не охвачены этой главой. Для получения дополнительной информации просмотрите методы `putFields()` и `writeFields()` класса `ObjectOutputStream` и метод `readFields()` класса `ObjectInputStream`. Посмотрите также примеры по дополнительным возможностям сериализации, приведенные в документации по Java SDK.

Сериализация апплетов

Особенно интересным применением сериализации объектов является сериализация апплетов (см. главу 15 «Апплеты»). При появлении Java 1.1 у HTML-тега <APPLET> появляется новый атрибут `OBJECT`, который можно использовать вместо атрибута `CODE` для задания файла сериализованного объекта вместо файла класса. Встретив такой тег <APPLET>, *визуализатор апплетов* или броузер создают апплет, десериализуя его.

Интересно это в силу того, что таким образом апплет может распространяться в заранее инициализированном состоянии. Код апплета даже может не содержать инициализирующий код. Для примера представим себе графическое средство разработки GUI, позволяющее программисту создавать GUI по технологии point-and-click («укажи-и-щелкни»). Такое средство разработки позволяет создать дерево AWT-компонентов в панели `Applet` и сериализовать апплет вместе со всеми компонентами GUI, которые он содержит. После десериализации апплет будет содержать полный GUI, несмотря на то что файл класса апплета не содержит никакого кода, создающего этот GUI.

Вы можете поэкспериментировать с сериализацией апплетов при помощи программы *appletviewer*. Запустите сначала апплет на *appletviewer* обычным способом. *appletviewer* загрузит апплет и запустит его методы `init()` и `start()`. Затем выберите в меню элемент **Stop**, чтобы ос-

тановить апплет. Теперь примените элемент меню **Save** для сериализации апплета в файл. По принятому соглашению вашему сериализованному апплету следует дать расширение *.ser*. Если апплет ссылается на какой-нибудь несериализуемый объект, сериализация может не удаться. Проблемы могут встретиться при сериализации апплета, использующего потоки исполнения (thread) или изображения (image).

Выполнив сериализацию апплета, создайте HTML-файл с подобным тегом <APPLET>:

```
<APPLET OBJECT="MyApplet.ser" WIDTH=400 HEIGHT=200></APPLET>
```

Наконец, примените *appletviewer* к этому новому HTML-файлу. Он должен десериализовать и отобразить апплет. Для апплета, созданного таким способом, метод апплета `init()` не вызывается (поскольку он вызывался до сериализации), но его метод `start()` вызывается (поскольку апплет должен был быть остановлен перед сериализацией).

Упражнения

9-1. Класс `java.util.Properties`, по существу, является хеш-таблицей, преобразующей строковые ключи в строковые значения. Он определяет метод `store()`, сохраняющий содержимое класса в байтовый поток, и метод `load()`, загружающий его оттуда. Эти методы, хотя и используют байтовый поток, сохраняют объект `Properties` в удобном для человеческого восприятия текстовом формате. Класс `Properties` наследует от интерфейса `Serializable` через свой родительский класс `java.util.Hashtable`. Определите подкласс `Properties`, реализующий методы `storeBinary()` и `loadBinary()`, использующие сериализацию объектов для сохранения и загрузки объекта `Properties` в двоичной форме. Вам может также понадобиться использовать в своих методах потоки `java.util.zip.GZIPOutputStream` и `java.util.zip.GZIPInputStream`, чтобы сделать ваш двоичный формат еще более компактным. Примените программу *serialver*, чтобы получить значение `serialVersionUID` для вашего класса.

9-2. Как отмечалось в предыдущем примере, объект `Properties` имеет методы `store()` и `load()`, позволяющие ему сохранять и восстанавливать свое состояние. Определите производный от `Externalizable` подкласс `Properties`, использующий методы `store()` и `load()` как основу для его методов `writeExternal()` и `readExternal()`. Чтобы все это заработало, метод `writeExternal()` должен определять количество байт, записанных методом `store()`, и записывать это значение первым, перед записываемыми байтами. Это позволит методу `readExternal()` узнать, когда перестать считывать. Включите также в используемый вами экстернализируемый формат данных номер версии.

II

Графика и пользовательский интерфейс

Часть II содержит примеры, использующие графический пользовательский интерфейс и графические возможности Java. Эти примеры соответствуют той части языка, которая была рассмотрена в книге «Java Foundation Classes in a Nutshell».

Глава 10. Графические интерфейсы пользователя (GUI)

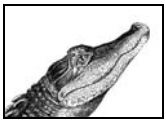
Глава 11. Графика

Глава 12. Печать

Глава 13. Передача данных

Глава 14. JavaBeans

Глава 15. Апплеты



Глава 10

Графические интерфейсы пользователя (GUI)

Графические интерфейсы пользователя, или GUI (Graphical User Interface), представляют собой превосходный пример модульности и многократного использования программного обеспечения. GUI почти всегда собираются из готовых строительных блоков, хранящихся в библиотеках. Motif-программистам в Unix-системах такие строительные блоки GUI известны как *widget*. Программистам под Windows они известны как *control* (элемент управления). В Java их называют общим термином *компонент* (*component*), поскольку все они являются подклассами `java.awt.Component`.[1]

В Java 1.0 и 1.1 стандартной библиотекой компонентов GUI был Abstract Windowing Toolkit (AWT) – пакет `java.awt` и его подпакеты. Помимо компонентов GUI пакет AWT включает в себя средства для рисования, выполнения перемещения данных по схеме cut-and-paste (вырезать и вставить) и прочих родственных операций. На большинстве платформ компоненты AWT реализованы с использованием систем GUI, свойственных самим операционным системам. То есть компоненты AWT реализованы на основе элементов управления Windows в операционных системах Windows, на основе графических компонентов Motif в Unix-системах и т. д. Такой стиль реализации привел к созданию средств разработки с наименьшим общим знаменателем, в результате чего интерфейс AWT API не так полно оснащен и завершен, как ему бы полагалось.

---

[1] За исключением компонентов AWT, связанных с меню. Они все являются подклассами `java.awt.MenuComponent`.

Java 1.2 вводит новую библиотеку компонентов GUI, известную как Swing. Swing состоит из пакета `javax.swing` и его подпакетов. В отличие от AWT, Swing обладает платформенно-независимой реализацией и полнофункциональным набором возможностей. Хотя Swing входит в основную часть платформы Java только начиная с Java 1.2, доступна версия Swing для работы с платформами Java 1.1. При создании GUI пакет Swing уже почти вытеснил AWT, поэтому в данной главе мы сосредоточимся на Swing.[1] Обратите внимание, что Swing определяет новый, более мощный набор компонентов GUI, но остается приверженцем той же базовой модели программирования GUI, что и AWT. Таким образом, научившись создавать GUI при помощи компонентов Swing, вы сможете делать то же самое с компонентами AWT.

Создание GUI на основе Java происходит в четыре основных этапа:

1. *Создание и конфигурирование компонентов*

   Компонент GUI создается, как и любой другой объект в Java, путем вызова его конструктора. Необходимо изучить документацию на индивидуальный компонент, чтобы узнать, каких аргументов ожидает конструктор. Например, для создания Swing-компонента `JButton` с надписью «Quit», следует просто написать:

   ```
   JButton quit = new JButton("Quit");
   ```

   Создав компонент, можно сконфигурировать его, задав одно или несколько его свойств. Например, чтобы задать шрифт, который должен использовать компонент `JButton`, можно написать:

   ```
   quit.setFont(new Font("sansserif", Font.BOLD, 18));
   ```

   И опять, следует познакомиться с документацией на используемый компонент, чтобы узнать, какие методы можно применять для его конфигурирования.

2. *Помещение компонента в контейнер*

   Все компоненты должны размещаться в *контейнере*. Все контейнеры в Java являются подклассами `java.awt.Container`. К обычно используемым контейнерам принадлежат классы `JFrame` и `JDialog`, представляющие окна верхнего уровня и диалоговые окна соответственно. Класс `java.applet.Applet`, представляющий собой базовый класс для апплетов, также является контейнером и, следовательно, может содержать и отображать компоненты GUI. Контейнер – это вид компонента, поэтому контейнеры могут быть помещены и часто помещаются в другие контейнеры. Контейнер `JPanel` часто используется следующим образом. При разработке GUI фактически создается иерархия контейнеров: окно верхнего уровня или апплет

[1] Исключение здесь представляют апплеты, которые по соображениям совместимости с большинством установленных на данный момент броузеров, не поддерживающих Swing, часто используют компоненты AWT.

содержат контейнеры, которые могут содержать другие контейнеры, которые в свою очередь содержат компоненты. Чтобы поместить компонент в контейнер, его просто передают методу `add()` контейнера. Например, кнопку `quit` можно поместить в контейнер «button box» посредством кода подобного следующему:

```
buttonbox.add(quit);
```

3. *Размещение или компоновка компонентов*

   Помимо определения того, какой компонент внутри какого контейнера находится, необходимо также задать положение и размер каждого компонента в его контейнере так, чтобы GUI хорошо смотрелся. Хотя можно включить положение и размер каждого компонента в код программы, гораздо чаще применяют объект `LayoutManager` для автоматического размещения компонентов в контейнере в соответствии с определенными правилами компоновки, задаваемыми конкретным, выбранным программистом объектом `LayoutManager`. Мы больше узнаем об управлении компоновкой далее в этой главе.

4. *Обработка событий, генерируемых компонентами*

   Описанных выше шагов достаточно для создания GUI, который хорошо смотрится на экране, но наш графический «интерфейс пользователя» не закончен, поскольку он еще не реагирует на действия пользователя. При взаимодействии пользователя с компонентами GUI при помощи клавиатуры или мыши эти компоненты генерируют, или активизируют, события. *Событие* (*event*) – это просто объект, содержащий информацию о действиях пользователя. Последним шагом при создании GUI является добавление *слушателей событий* (*event listeners*) – объектов, которым посылаются уведомления о том, что произошло событие, и которые адекватно отвечают на это событие. Например, кнопке **Quit** нужен слушатель, который вызовет выход из приложения.

Мы рассмотрим каждую из этих тем подробно в нескольких первых разделах данной главы. Затем мы перейдем к более специальным примерам GUI, освещающим отдельные компоненты Swing или особые приемы программирования GUI. При изучении примеров вам, возможно, понадобится обратиться к справочным материалам по AWT и Swing. Одним из таких справочников служит книга «Java Foundation Classes in a Nutshell». Настоящую главу полезно читать вместе с главами 2 и 3 этой книги.

Компоненты

Как мы только что обсудили, первым шагом при создании GUI является создание и конфигурирование составляющих его компонентов. Для этого вам необходимо знать множество доступных компонентов и их методы. Глава 2 книги «Java Foundation Classes in a Nutshell» содер-

жит таблицы, в которых перечисляются доступные компоненты AWT и Swing. Можно также получить эту информацию из списка классов, входящих в пакеты `java.awt` и `javax.swing`; особенно удобно искать компоненты Swing, поскольку все их имена начинаются с буквы J.

Каждый компонент определяет набор свойств, которые можно использовать для его конфигурирования. *Свойство* (*property*) – это снабженный именем атрибут компонента, значение которого можно установить. Типичные свойства компонентов имеют такие имена: `font`, `background` и `alignment`. Значение свойства устанавливается при помощи *метода установки* (*setter*) и опрашивается при помощи *метода опроса* (*getter*). Методы получения и установки свойств в совокупности называются *методами доступа к свойствам* (*accessor methods*), их имена обычно начинаются со слов «set» и «get». Понятию свойства компонента дается формальное определение в спецификациях JavaBeans; мы узнаем о них больше в главе 14 «JavaBeans». Однако пока нам будет достаточно следующего неформального определения: компоненты можно конфигурировать, вызывая различные методы, имена которых начинаются с «set». Запомним, что компоненты наследуют множество методов от своих родительских классов, в особенности от `java.awt.Component` и `javax.swing.JComponent`, поэтому только из того, что компонент сам не определяет какой-то метод установки, еще не следует, что компонент не поддерживает соответствующее свойство.

Пример 10.1 – это листинг программы *ShowComponent.java*. Эта программа предоставляет простой способ поэкспериментировать с компонентами AWT и Swing и их свойствами. Ее можно вызвать с одним или несколькими именами компонентов в командной строке. За каждым именем класса может следовать нуль или большее число спецификаций свойств в форме *имя=значение*. Программа создает экземпляр для каждого названного класса и конфигурирует его, присваивая упомянутым свойствам заданные значения. Затем программа отображает компоненты, используя Swing-контейнер `JTabbedPane` в окне `JFrame`. Например, для создания окна, показанного на рис. 10.1, я вызывал

Рис. 10.1. *Компоненты, отображенные программой ShowComponent*

ShowComponent следующим образом (это одна длинная команда, свернутая в несколько строк):

```
% java com.davidflanagan.examples.gui.ShowComponent javax.swing.JButton
'text=Hello World!' font=helvetica-bold-48 javax.swing.JRadioButton
'text=pick me' java.awt.Button label=Hello javax.swing.JSlider
```

Одной из мощных возможностей библиотеки компонентов Swing является использование ею архитектуры подключаемых стилей, позволяющее изменять внешний вид GUI в целом. Принимаемый по умолчанию внешний вид (стиль «Metal») – один и тот же для всех платформ. Однако программа может выбрать любой установленный стиль, включая стиль, симулирующий внешний вид приложений родной операционной системы. Одной из интересных составляющих программы ShowComponent является раскрывающееся меню **Look and Feel**. Оно позволяет выбрать любой из установленных стилей.[1]

Многие примеры в этой главе имеют форму подклассов компонентов. В дополнение к использованию программы ShowComponent как средства для экспериментов с готовыми компонентами Swing и AWT мы будем применять ее для просмотра других примеров. Но ShowComponent – это не только служебная программа; она сама по себе является интересным примером. Она демонстрирует основные шаги при создании простого Swing GUI, включающего в себя компоненты JFrame, JMenuBar и JTabbedPane. Изучив метод main(), вы увидите, как окно создается, конфигурируется и выводится на экран. Метод createPlafMenu() демонстрирует создание и конфигурирование раскрывающегося меню. Остальную часть примера 10.1 составляет метод getComponentsFromArgs(), изощренный пример использования механизма отражения Java (а также интроспекции JavaBeans) для работы с GUI. Этот код интересен, хотя и не напрямую связан с темой этой главы.

Пример 10.1. ShowComponent.java

```
package com.davidflanagan.examples.gui;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.beans.*;
import java.lang.reflect.*;
import java.util.Vector;
```

[1] Хотя стиль Windows входит в стандартную поставку Java, по юридическим соображениям его использование недопустимо ни с какими операционными системами, кроме Windows. Поэтому, хотя такой вариант выбора есть в меню, вы не сможете его применить, если используете операционную систему не от Microsoft. Аналогично реализация, эмулирующая стиль MacOS, доступна на Apple, а Java-реализация нового стиля Apple – «Aqua» – ожидается вскоре.

```
/**
 * Этот класс является программой, которая использует отражение и JavaBeans-
 * интроспекцию для создания набора заданных компонентов, установки заданных
 * свойств этих компонентов и их отображения на экране. Она позволяет
 * пользователю увидеть компоненты с применением любого установленного стиля.
 * Она создана как простое средство для экспериментирования с компонентами
 * AWT и Swing и для просмотра других примеров, предложенных в этой главе.
 * Она демонстрирует также окна (frames), меню и компонент JTabbedPane.
 **/
public class ShowComponent {
  // Главная программа
  public static void main(String[] args) {
    // Обрабатываем командную строку для получения отображаемых компонентов
    Vector components = getComponentsFromArgs(args);

    // Создаем окно для отображения в нем компонентов
    JFrame frame = new JFrame("ShowComponent");

    // На закрытие окна реагируем выходом из VM
    frame.addWindowListener(new WindowAdapter() { // Безымянный внутренний
                                                  // класс
        public void windowClosing(WindowEvent e) { System.exit(0); }
      });

    // Устанавливаем меню, позволяющее пользователю выбрать
    // стиль компонента в списке установленных PLAF
    JMenuBar menubar = new JMenuBar();      // Создаем панель меню
    frame.setJMenuBar(menubar);             // Приказываем окну отобразить ее
    JMenu plafmenu = createPlafMenu(frame); // Создаем меню
    menubar.add(plafmenu);                  // Помещаем меню в полосу меню

    // Создаем JTabbedPane для отображения всех компонентов
    JTabbedPane pane = new JTabbedPane();

    // Теперь помещаем каждый компонент как вкладку на панель со вкладками
    // Используем сокращенное имя класса компонента как текст на ярлыке вкладки
     for(int i = 0; i < components.size(); i++) {
      Component c = (Component)components.elementAt(i);
      String classname = c.getClass().getName();
      String tabname = classname.substring(classname.lastIndexOf('.')+1);
      pane.addTab(tabname, c);
    }

    // Помещаем панель с вкладками в окно. Обратите внимание на вызов
    // getContentPane(). Это обязательно для JFrame, но не необходимо
    // для большинства компонентов Swing
    frame.getContentPane().add(pane);

    // Устанавливаем размер окна и показываем его
    frame.pack();       // Делаем размер окна таким, чтобы в нем
                        // поместились все его элементы
    frame.setVisible(true);  // Делаем окно видимым на экране

    // Здесь происходит выход из метода main(), но Java VM продолжает
```

```
    // работать, т. к. все AWT-программы автоматически запускают поток
    // (thread) обработки событий.
  }

  /**
   * Этот статический метод опрашивает систему, желая узнать, какие
   * подключаемые стили (Pluggable Look-and-Feel, PLAF) доступны.
   * Затем он создает компонент JMenu, перечисляющий все стили по имени
   * и позволяющий пользователю выбрать один из них при помощи компонентов
   * JRadioButtonMenuItem. Когда пользователь сделает выбор, выбранный
   * элемент меню обходит иерархию компонентов, приказывая каждому
   * компоненту применить новый PLAF.
   **/
  public static JMenu createPlafMenu(final JFrame frame) {
    // Создаем меню
    JMenu plafmenu = new JMenu("Look and Feel");

    // Создаем объект для взаимоисключающего выбора на основе
    // переключателей (radio button)
    ButtonGroup radiogroup = new ButtonGroup();

    // Просматриваем доступные стили
    UIManager.LookAndFeelInfo[] plafs =
      UIManager.getInstalledLookAndFeels();

    // Проходим стили в цикле, помещая в меню элемент для каждого из них
    for(int i = 0; i < plafs.length; i++) {
      String plafName = plafs[i].getName();
      final String plafClassName = plafs[i].getClassName();

      // Создаем элемент меню
      JMenuItem item = plafmenu.add(new JRadioButtonMenuItem(plafName));

      // Сообщаем элементу меню, что нужно делать, если его выберут
      item.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
            try {
              // Устанавливаем новый стиль
              UIManager.setLookAndFeel(plafClassName);
              // Приказываем каждому компоненту изменить свой стиль
              SwingUtilities.updateComponentTreeUI(frame);
              // Приказываем окну изменить его размер в соответствии
              // с новыми желаемыми размерами его элементов
              frame.pack();
            }
            catch(Exception ex) { System.err.println(ex); }
          }

        });

      // Только один элемент меню может быть выбран в каждый момент
      radiogroup.add(item);
    }
    return plafmenu;
  }
```

```
/**
 * Этот метод в цикле проходит аргументы, заданные в командной строке,
 * отыскивая имена классов компонентов для создания компонентов и установки
 * их свойств, заданных в форме name=value. Этот метод демонстрирует
 * отражение и интроспекцию JavaBeans так, как они могут быть применены
 * к динамически создаваемому GUI
 **/
public static Vector getComponentsFromArgs(String[] args) {
  Vector components = new Vector();      // Список возвращаемых компонентов
  Component component = null;            // Текущий компонент
  PropertyDescriptor[] properties = null; // Свойства компонента
  Object[] methodArgs = new Object[1];    // Это нам пригодится ниже

 nextarg: // Это цикл с меткой
  for(int i = 0; i < args.length; i++) { // Цикл по аргументам
    // Если аргумент не содержит знак равенства, значит, это
    // имя класса компонента, иначе это спецификация свойства
    int equalsPos = args[i].indexOf('=');
    if (equalsPos == -1) { // Это имя компонента
      try {
        // Загружаем заданный класс
        Class componentClass = Class.forName(args[i]);
        // Создаем его экземпляр для создания экземпляра компонента
        component = (Component)componentClass.newInstance();
        // Применяем JavaBeans для изучения внутреннего устройства
        // компонента. И получаем список свойств, которые он поддерживает
        BeanInfo componentBeanInfo =
          Introspector.getBeanInfo(componentClass);
        properties = componentBeanInfo.getPropertyDescriptors();
      }
      catch(Exception e) {
       // При неудаче на каком-то шаге печатаем сообщение об ошибке и выходим
        System.out.println("Невозможно загрузить, создать экземпляр " +
                  " или провести интроспекцию: " + args[i]);
        System.exit(1);
      }

      // Если все прошло успешно, сохраняем компонент в векторе
      components.addElement(component);
    }
    else { // Текущий аргумент – задание свойства в форме name=value
      String name =args[i].substring(0, equalsPos); // имя свойства
      String value =args[i].substring(equalsPos+1); // значение свойства

      // Если у нас нет компонента, чтобы установить свойство, пропускаем!
      if (component == null) continue nextarg;

      // Теперь просматриваем дескрипторы свойств этого компонента,
      // чтобы найти свойство с тем же именем.
      for(int p = 0; p < properties.length; p++) {
        if (properties[p].getName().equals(name)) {
          // Отлично, мы нашли свойство с нужным именем.
          // Теперь получаем его тип и метод установки
```

```
Class type = properties[p].getPropertyType();
Method setter = properties[p].getWriteMethod();

// Проверяем, не является ли свойство доступным только для чтения!
if (setter == null) {
  System.err.println("Свойство " + name+
            " только для чтения");
  continue nextarg; // переходим к следующему аргументу
}

// Пытаемся привести заданное значение свойства к правильному типу
// Здесь мы поддерживаем маленький набор обычных типов свойств
// Сохраняем преобразованное значение в Object[], так что
// теперь его будет удобно передать методу установки
try {
  if (type == String.class) {       // преобразование не нужно
    methodArgs[0] = value;
  }
  else if (type == int.class) {     // Строку в целое
    methodArgs[0] = Integer.valueOf(value);
  }
  else if (type == boolean.class) { // в логическое
    methodArgs[0] = Boolean.valueOf(value);
  }
  else if (type == Color.class) {   // в Color (цвет)
    methodArgs[0] = Color.decode(value);
  }
  else if (type == Font.class) {    // Строку в Font (шрифт)
    methodArgs[0] = Font.decode(value);
  }
  else {
    // Если не можем преобразовать, пропускаем свойство
    System.err.println("Свойство " + name +
              " неподдерживаемого типа" +
              type.getName());
    continue nextarg;
  }
}
catch (Exception e) {
  // Если преобразование потерпело неудачу, переходим
  // к следующему аргументу
  System.err.println("Невозможно привести `" + value +
            "' к типу " + type.getName() +
            " для свойства " + name);
  continue nextarg;
}

// Наконец, применяем отражение, чтобы вызвать метод
// установки для созданного нами компонента,
// и передаем ему преобразованное значение свойства.
try { setter.invoke(component, methodArgs); }
catch (Exception e) {
```

```
                System.err.println("Невозможно установить свойство: " + name);
              }

              // Теперь переходим к следующему заданному в командной
              // строке аргументу
              continue nextarg;
            }
          }

          // Если мы оказались здесь, значит, не нашли заданное свойство
          System.err.println("Предупреждение: нет такого свойства: " + name);
        }
      }

      return components;
    }
}
```

Контейнеры

Второй шаг при создании GUI – это помещение созданных и сконфигурированных компонентов в подходящие контейнеры. Глава 2 книги «Java Foundation Classes in a Nutshell» содержит таблицы, где перечислены классы контейнеров, имеющиеся в пакетах AWT и Swing. Многие из этих классов контейнеров имеют специальное назначение. Например, `JFrame` – это окно верхнего уровня, а `JTabbedPane` отображает компоненты, которые он содержит, на отдельных панелях со вкладками. Пример 10.1 демонстрирует применение этих контейнеров. Но Swing и AWT содержат также общие классы контейнеров, такие как `JPanel`.

Пример 10.2 – это листинг программы *Containers.java*. Этот класс является подклассом `JPanel`. Его метод-конструктор создает множество других вложенных экземпляров `JPanel`, равно как и множество объектов `JButton`, содержащихся в этих классах `JPanel`. Пример 10.2 иллюстрирует понятие иерархии контейнеров в GUI, применяя цвет для обозначения глубины вложения в этой иерархии. На рис. 10.2 показано, как выглядит класс `Containers` при отображении программой `ShowCompo-`

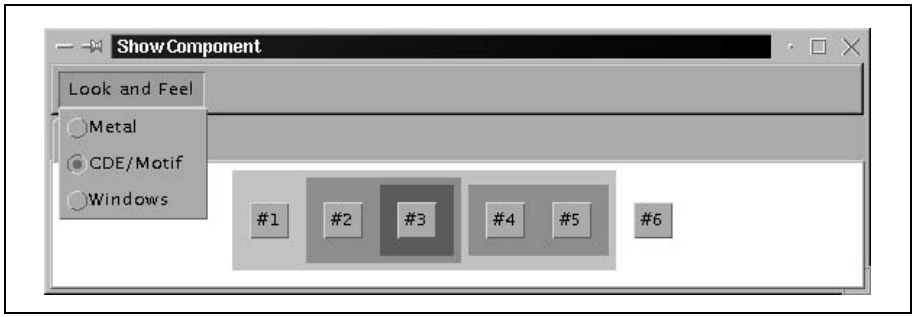


***Рис. 10.2.** Вложенные контейнеры*

nent, запущенной следующим образом (еще одна длинная команда, разбитая на две строки):

```
% java com.davidflanagan.examples.gui.ShowComponent \
com.davidflanagan.examples.gui.Containers
```

Для разнообразия на рис. 10.2 используется стиль Motif и показано содержимое меню **Look and Feel**.

Пример 10.2. Containers.java

```
package com.davidflanagan.examples.gui;
import javax.swing.*;
import java.awt.*;

/**
 * Подкласс компонента, демонстрирующий вложенные контейнеры и компоненты.
 * Он создает приведенную ниже иерархию и использует различные цвета
 * для различения глубины вложенности контейнеров
 *
 *  containers---panel1----button1
 *           |       |---panel2----button2
 *           |       |        |----panel3----button3
 *           |       |------panel4----button4
 *           |                  |----button5
 *           |---button6
 */
public class Containers extends JPanel {
  public Containers() {
    this.setBackground(Color.white);      // Этот компонент белый
    this.setFont(new Font("Dialog", Font.BOLD, 24));

    JPanel p1 = new JPanel();
    p1.setBackground(new Color(200, 200, 200)); // Panel1 темнее
    this.add(p1);                 // p1 содержится в этом компоненте
    p1.add(new JButton("#1"));    // Button1 содержится в p1

    JPanel p2 = new JPanel();
    p2.setBackground(new Color(150, 150, 150)); // p2 темнее, чем p2
    p1.add(p2);                   // p2 содержится в p1
    p2.add(new JButton("#2"));    // Button2 содержится в p2

    JPanel p3 = new JPanel();
    p3.setBackground(new Color(100, 100, 100)); // p3 темнее, чем p2
    p2.add(p3);                   // p3 содержится в p2
    p3.add(new JButton("#3"));    // Button3 содержится в p3

    JPanel p4 = new JPanel();
    p4.setBackground(new Color(150, 150, 150)); // p4 темнее, чем p1
    p1.add(p4);                   // p4 содержится в p1
    p4.add(new JButton("#4"));    // Button4 содержится в p4
    p4.add(new JButton("#5"));    // Button5 тоже содержится в p4

    this.add(new JButton("#6")); // Button6 содержится в этом компоненте
  }
}
```

Управление компоновкой

Создав компоненты и поместив их в контейнеры, на следующем шаге нужно расположить их в контейнере. Эта деятельность называется *управлением компоновкой* и почти всегда осуществляется специальным объектом – *менеджером компоновки* (*layout manager*). Менеджеры компоновки являются реализациями интерфейса `java.awt.LayoutManager` или производного от него интерфейса `LayoutManager2`. Каждая конкретная реализация `LayoutManager` придерживается собственных правил компоновки и автоматически располагает компоненты в контейнере в соответствии с этими правилами. В следующих разделах демонстрируется применение всех менеджеров компоновки AWT и Swing. Заметим, что `BoxLayout` – это единственный менеджер компоновки, определенный в Swing. Хотя Swing определяет много новых компонентов, Swing GUI обычно использует менеджеры компоновки AWT.

Менеджер компоновки создается, как любой другой объект. Разные классы менеджеров компоновки принимают разные аргументы своих конструкторов для задания параметров их правил компоновки. После создания менеджера компоновки вызывать его методы приходится редко. Вместо этого объект менеджера компоновки передается методу `setLayout()` того контейнера, которым нужно управлять; контейнер при необходимости сам вызывает разные методы `LayoutManager`. Обычно, создав менеджер компоновки, о нем можно благополучно забыть.

Как вы это увидите из следующих разделов, большая часть готовых менеджеров компоновки из AWT следуют весьма простым принципам компоновки, которые сами по себе кажутся не очень полезными. Однако их сила обнаруживается, когда они применяются совместно. Например, можно использовать `GridLayout` для размещения 10 кнопок в контейнере в два столбца, а затем применить `BorderLayout`, чтобы поместить эти два столбца у левого края другого контейнера.

Следующие разделы демонстрируют все основные менеджеры компоновки, используя короткие примеры и фрагменты экрана с изображением созданной в примере компоновки. Рисунки созданы с использованием класса `ShowComponent` из примера 10.2; вы можете воспользоваться этой программой, чтобы самостоятельно поэкспериментировать с примерами. Обратите особенное внимание на то, как изменяется расположение компонентов при изменении размеров окна.

FlowLayout

Менеджер компоновки `FlowLayout` располагает свои элементы, как слова на странице: слева направо и сверху вниз. Когда в текущей строке не остается места для очередного компонента, `FlowLayout` переходит на новую строку. При создании `FlowLayout` можно задать выравнивание по левому или правому краю или по центру. Можно задать также расстo-

яние, которое менеджер компоновки будет оставлять между компонентами. FlowLayout не стремится «заполнить» своими компонентами весь контейнер; он оставляет каждому компоненту его первоначальный размер. Если остается свободное место, FlowLayout оставляет его пустым. Если в контейнере не хватает места, некоторые компоненты просто не будут видны. Обратите внимание на то, что FlowLayout – это используемый по умолчанию менеджер компоновки для контейнеров JPanel. Если не задан другой менеджер компоновки, панель применит FlowLayout, который будет центрировать свои строки и оставит по пять пикселов между компонентами – и по горизонтали, и по вертикали.

Пример 10.3 – это короткая программа, располагающая кнопки с применением менеджера компоновки FlowLayout; на рис. 10.3 показан результат.

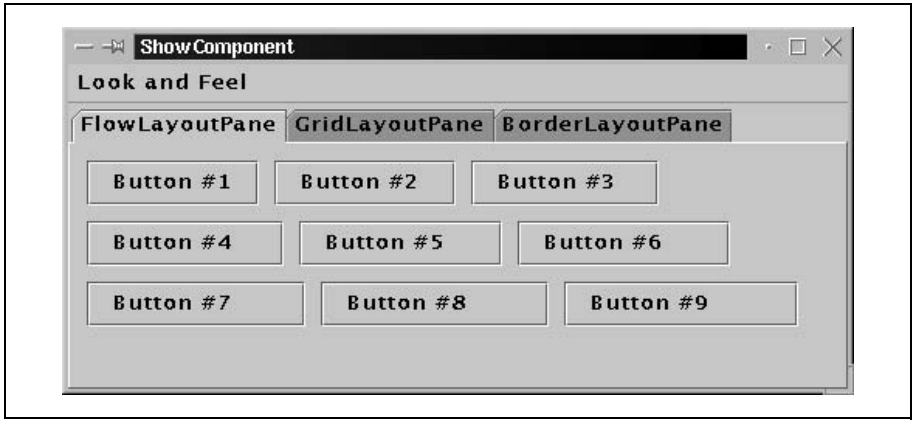


Рис. 10.3. *Компоненты, скомпонованные программой FlowLayout*

Пример 10.3. FlowLayoutPane.java

```
package com.davidflanagan.examples.gui;
import java.awt.*;
import javax.swing.*;

public class FlowLayoutPane extends JPanel {
  public FlowLayoutPane() {
    // Применяем менеджер компоновки FlowLayout. Строки выровнены по левому
    // краю. Между компонентами - и по горизонтали, и по вертикали -
    // оставляется по 10 пикселов.
    this.setLayout(new FlowLayout(FlowLayout.LEFT, 10, 10));

    // Добавляем несколько кнопок, чтобы продемонстрировать компоновку.
    String spaces = ""; // Применяется, чтобы сделать кнопки разными
    for(int i = 1; i <= 9; i++) {
      this.add(new JButton("Button #" + i + spaces));
      spaces += " ";
    }
```

```
        // Сами задаем принимаемый по умолчанию размер
        this.setPreferredSize(new Dimension(500, 200));
    }
}
```

GridLayout

`GridLayout` — это строгий менеджер компоновки. Он размещает компоненты слева направо и сверху вниз в клетках равномерной решетки. При создании `GridLayout` можно задавать число строк и столбцов решетки, равно как горизонтальные и вертикальные интервалы, которые менеджер `GridLayout` должен оставлять между компонентами. Обычно задают только число строк или столбцов, оставляя другую размерность равной `0`. Это позволяет `GridLayout` подобрать требуемое число строк или столбцов на основе числа компонентов. `GridLayout` не учитывает первоначальные размеры своих компонентов. Вместо этого он делит размер контейнера на заданное число строк и столбцов одинакового размера и делает все компоненты одинаковыми по размеру.

Пример 10.4 показывает короткую программу, которая размещает кнопки в решетку, используя менеджер компоновки `GridLayout`. На рис. 10.4 показан полученный вывод.

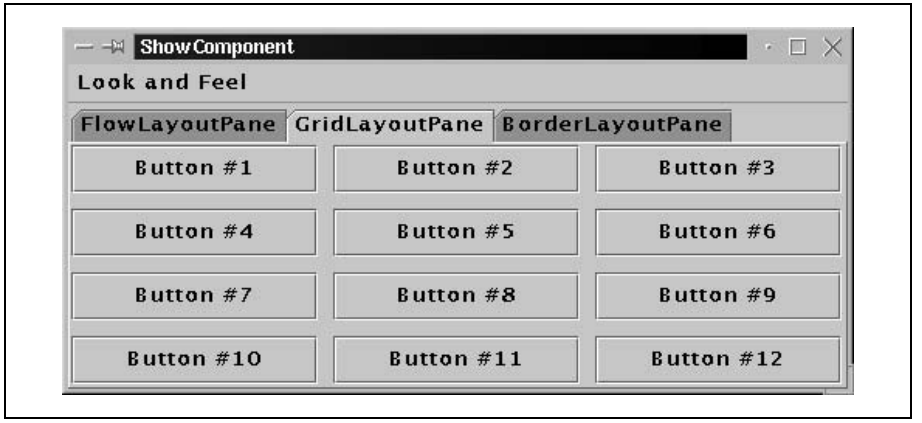


***Рис. 10.4.** Компоненты, скомпонованные программой GridLayout*

Пример 10.4. GridLayoutPane.java

```
package com.davidflanagan.examples.gui;
import java.awt.*;
import javax.swing.*;

public class GridLayoutPane extends JPanel {
  public GridLayoutPane() {
    // Располагаем компоненты в решетке с тремя столбцами и числом строк,
    // зависящим от числа компонентов. Оставляем между компонентами
    // 10 пикселов по горизонтали и по вертикали
```

```
        this.setLayout(new GridLayout(0, 3, 10, 10));
        // Помещаем компоненты в контейнер
        for(int i = 1; i <= 12; i++) this.add(new JButton("Button #" + i));
      }
    }
```

BorderLayout

Менеджер компоновки BorderLayout размещает в контейнере не более пяти компонентов. Четыре компонента располагаются по сторонам контейнера, а один помещается в центре. При помещении компонента в контейнер, которым управляет BorderLayout, нужно указать, где компоненты следует разместить. Это осуществляется посредством двухаргументной версии метода add(), которой в качестве второго аргумента передается одна из констант NORTH, EAST, SOUTH, WEST или CENTER, определенных BorderLayout. Эти константы называются *ограничителями компоновки*; они часто будут встречаться примерно в таком виде:

```
this.add(b, BorderLayout.SOUTH);
```

Следует помнить, что BorderLayout может поместить только один компонент в каждую из этих позиций.

BorderLayout не полностью учитывает первоначальные размеры компонентов, которыми он управляет. Компоненты, которым предписано расположиться по сторонам NORTH и SOUTH, принимают ширину контейнера и сохраняют свою первоначальную высоту. Компоненты EAST и WEST получают высоту контейнера (за вычетом высот верхнего и нижнего компонентов, если таковые имеются) и сохраняют свою первоначальную ширину. Компонент CENTER занимает все оставшееся место в центральной части контейнера – после определения заданного числа пикселов для отступов по горизонтали и по вертикали. Нет необходимости задавать все пять элементов. К примеру, класс BorderLayout часто используется для размещения компонента фиксированного размера (такого как JToolBar) у одного края контейнера, а компонента переменного размера (такого как JTextArea) – в центре, на всем оставшемся пространстве.

BorderLayout – это принимаемый по умолчанию менеджер компоновки для контейнеров панелей содержания JFrame и JDialog. Если менеджер компоновки для этих панелей не задан, они используют BorderLayout, сконфигурированный так, чтобы он не оставлял никаких пустот между компонентами.

В примере 10.5 приводится программа, которая размещает пять кнопок при помощи менеджера компоновки BorderLayout. На рис. 10.5 показан полученный вывод.

Пример 10.5. BorderLayoutPane.java

```
package com.davidflanagan.examples.gui;
import java.awt.*;
```

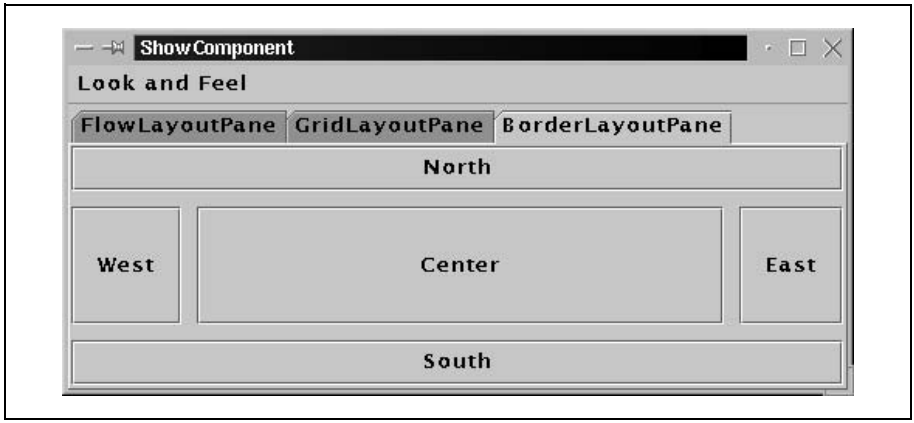


Рис. 10.5. *Компоненты, скомпонованные программой BorderLayout*

```
import javax.swing.*;

public class BorderLayoutPane extends JPanel {
  String[] borders = {"North", "East", "South", "West", "Center"};
  public BorderLayoutPane() {
    // Применяем BorderLayout с промежутками в 10 пикселов между компонентами
    this.setLayout(new BorderLayout(10, 10));
    for(int i = 0; i < 5; i++) {          // Помещаем компоненты в панель
      this.add(new JButton(borders[i]),   // Добавляем компонент
           borders[i]);                   // с этим ограничителем
    }
  }
}
```

Box и BoxLayout

`javax.swing.BoxLayout` — это простой, но гибкий менеджер компоновки. Он располагает свои компоненты в строку или в столбец. Контейнер `javax.swing.Box` использует `BoxLayout`; гораздо чаще работают с классом `Box`, чем напрямую применяют `BoxLayout`. Гибкость контейнеров `Box` проявляется в возможности добавления эластичных (*склейка, glue*) и жестких (*распорка, strut*) промежутков между компонентами. Класс `Box` определяет статические методы, которые делают его весьма удобным при создании строк, столбцов, склеек и распорок.

В примере 10.6 для демонстрации способностей `BoxLayout` создаются несколько контейнеров `Box`. Различные прямоугольники (box) сами компонуются с применением `BorderLayout`. Результат работы программы показан на рис. 10.6. Чтобы вы не заскучали, пример 10.6 демонстрирует также применение Swing-рамок для создания полей и украшений вокруг некоторых контейнеров. Обратите внимание на то, что эти рамки могут быть нарисованы вокруг любого компонента или контейнера Swing; они входят в пакет `javax.swing.border` и никак не связаны с

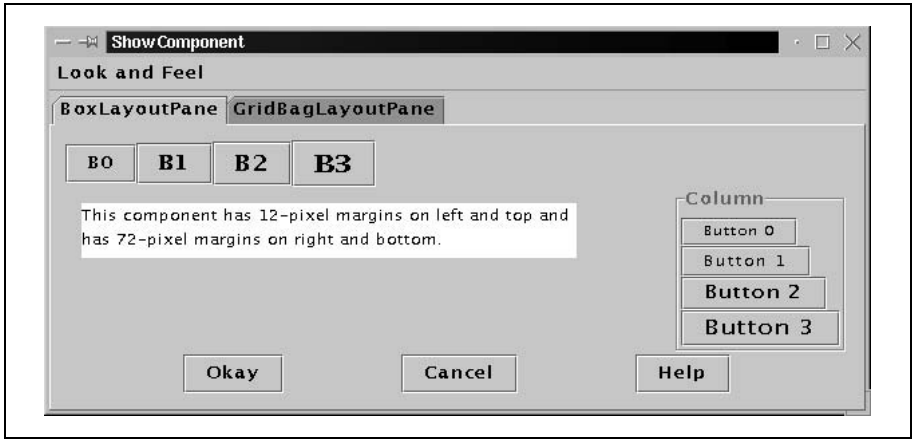


Рис. 10.6. *Компоненты, скомпонованные программой BoxLayout*

менеджером компоновки `BorderLayout`. См. класс `javax.swing.border.Border` и метод `setBorder()` класса `JComponent`.

Пример 10.6. BoxLayoutPane.java

```
package com.davidflanagan.examples.gui;
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

public class BoxLayoutPane extends JPanel {
  public BoxLayoutPane() {
    // Применяем менеджер компоновки BorderLayout для размещения
    // разных компонентов Box
    this.setLayout(new BorderLayout());

    // Задаем поля для всей панели, помещая в нее пустую рамку
    // Это также можно было бы сделать, заместив getInsets()
    this.setBorder(new EmptyBorder(10, 10, 10, 10));

    // Размещаем вдоль верхнего края панели простой ряд кнопок
    Box row = Box.createHorizontalBox();
    for(int i = 0; i < 4; i++) {
      JButton b = new JButton("B" + i);
      b.setFont(new Font("serif", Font.BOLD, 12+i*2));
      row.add(b);
    }
    this.add(row, BorderLayout.NORTH);

    // Размещаем простой столбец кнопок вдоль правого края панели
    // Применяем BoxLayout для другого Swing-контейнера
    // Создаем рамку вокруг столбца: это не получилось бы с классом Box
    JPanel col = new JPanel();
    col.setLayout(new BoxLayout(col, BoxLayout.Y_AXIS));
    col.setBorder(new TitledBorder(new EtchedBorder(), "Column"));
```

```
        for(int i = 0; i < 4; i++) {
          JButton b = new JButton("Button " + i);
          b.setFont(new Font("sansserif", Font.BOLD, 10+i*2));
          col.add(b);
        }
        this.add(col, BorderLayout.EAST); // Размещаем столбец в панели справа

        // Размещаем прямоугольник с кнопками вдоль нижнего края панели.
        // Применяем «Glue» для равномерного распределения кнопок
        Box buttonbox = Box.createHorizontalBox();
        buttonbox.add(Box.createHorizontalGlue());   // эластичный промежуток
        buttonbox.add(new JButton("Okay"));
        buttonbox.add(Box.createHorizontalGlue());   // эластичный промежуток
        buttonbox.add(new JButton("Cancel"));
        buttonbox.add(Box.createHorizontalGlue());   // эластичный промежуток
        buttonbox.add(new JButton("Help"));
        buttonbox.add(Box.createHorizontalGlue());   // эластичный промежуток
        this.add(buttonbox, BorderLayout.SOUTH);

        // Создаем, чтобы отобразить в центре панели
        JTextArea textarea = new JTextArea();
        textarea.setText("This component has 12-pixel margins on left and top"+
                 " and has 72-pixel margins on right and bottom.");
        textarea.setLineWrap(true);
        textarea.setWrapStyleWord(true);

        // Используем объекты Box, чтобы окружить JTextArea нестандартными полями
        // Во-первых, создаем столбец из трех элементов. Первый и последний будут
        // жесткими промежутками. Средний элемент будет текстовой областью
        Box fixedcol = Box.createVerticalBox();
        fixedcol.add(Box.createVerticalStrut(12)); // 12 жестких пикселов
        fixedcol.add(textarea);     // Компонент заполняет остальное
        fixedcol.add(Box.createVerticalStrut(72)); // 72 жестких пиксела

        // Теперь создаем строку. Помещаем распорки слева и справа,
        // а определенный выше столбец вставляем посередине.
        Box fixedrow = Box.createHorizontalBox();
        fixedrow.add(Box.createHorizontalStrut(12));
        fixedrow.add(fixedcol);
        fixedrow.add(Box.createHorizontalStrut(72));

        // Теперь помещаем в панель область JtextArea, заключенную в столбец
        // и в строку
        this.add(fixedrow, BorderLayout.CENTER);
      }
    }
```

GridBagLayout

`GridBagLayout` — **самый гибкий и мощный из менеджеров компоновки AWT, но он также и самый сложный, а порой вызывает разочарование. Он располагает компоненты в соответствии с множеством ограничителей, которые хранятся в объекте** `GridBagConstraints`**. В Java 1.1 и**

более поздних версиях объект GridBagConstraints передается вместе с добавляемым компонентом методу add() в качестве второго аргумента. Однако в Java 1.0 объект ограничений для компонента задается путем вызова метода setConstraints() самого класса GridBagLayout.

Основной принцип компоновки GridBagLayout состоит в размещении компонентов по заданным позициям в решетке. Решетка может иметь произвольный размер, а строки и столбцы решетки могут быть произвольной высоты и ширины. Помещаемый на эту решетку компонент может занимать больше одной строки или одного столбца. Поля gridx и gridy объекта GridBagConstraints задают положение компонента в решетке, а поля gridwidth и gridheight задают число строк и столбцов, соответственно, которые компонент занимает в решетке. Поле insets задает пустое пространство, которое должно быть оставлено вокруг каждого отдельного компонента, а поле fill указывает, увеличится ли компонент и, если да, то как, в случае, когда для него останется пространства больше, чем нужно для его собственных, принимаемых по умолчанию размеров. Поле anchor указывает, как компонент следует позиционировать, когда имеется больше места, чем он занимает. GridBagConstraints определяет множество констант, приемлемых в качестве значений для этих двух полей. Наконец, поля weightx и weighty указывают, как должно распределяться между компонентами свободное пространство, образующееся при изменении размеров контейнера. За более подробной информацией по GridBagConstraints обратитесь к справочным материалам.

В примере 10.7 показана короткая программа, которая использует менеджер компоновки GridBagLayout для создания компоновки, показанной на рис. 10.7. Обратите внимание на то, что программа многократно использует один объект GridBagConstraints, что совершенно законно.

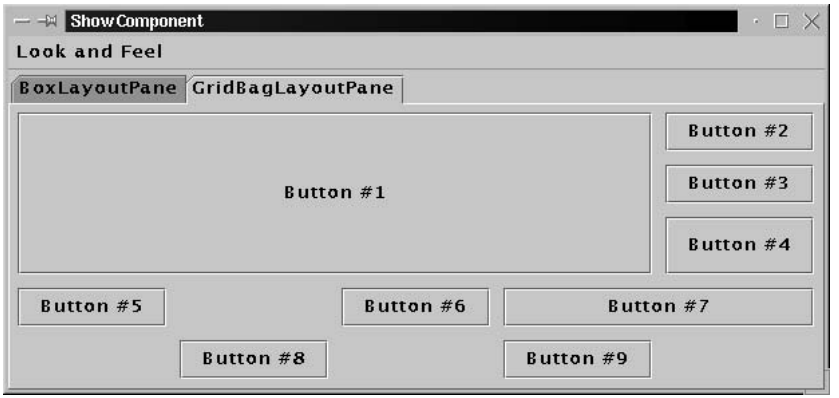


Рис. 10.7. *Компоненты, скомпонованные программой GridBagLayout*

Пример 10.7. GridBagLayoutPane.java

```
package com.davidflanagan.examples.gui;
import java.awt.*;
import javax.swing.*;

public class GridBagLayoutPane extends JPanel {
  public GridBagLayoutPane() {
    // Создаем и назначаем менеджер компоновки
    this.setLayout(new GridBagLayout());

    // Создаем объект ограничений и задаем некоторые принимаемые
    // по умолчанию значения
    GridBagConstraints c = new GridBagConstraints();
    c.fill = GridBagConstraints.BOTH; // компоненты растут
                                      // по обеим размерностям
    c.insets = new Insets(5,5,5,5);   // поля в 5 пикселов со всех сторон

    // Создаем и помещаем в контейнер группу кнопок, задавая для каждой разные
    // размеры и положение в решетке. Полагаем для первой кнопки коэффициент
    // чувствительности к размеру окна равным 1.0, а для других равным 0.0.
    // Первая кнопка будет занимать все освободившееся пространство.
    c.gridx = 0; c.gridy = 0; c.gridwidth = 4; c.gridheight=4;
    c.weightx = c.weighty = 1.0;
    this.add(new JButton("Button #1"), c);

    c.gridx = 4; c.gridy = 0; c.gridwidth = 1; c.gridheight=1;
    c.weightx = c.weighty = 0.0;
    this.add(new JButton("Button #2"), c);

    c.gridx = 4; c.gridy = 1; c.gridwidth = 1; c.gridheight=1;
    this.add(new JButton("Button #3"), c);

    c.gridx = 4; c.gridy = 2; c.gridwidth = 1; c.gridheight=2;
    this.add(new JButton("Button #4"), c);

    c.gridx = 0; c.gridy = 4; c.gridwidth = 1; c.gridheight=1;
    this.add(new JButton("Button #5"), c);

    c.gridx = 2; c.gridy = 4; c.gridwidth = 1; c.gridheight=1;
    this.add(new JButton("Button #6"), c);

    c.gridx = 3; c.gridy = 4; c.gridwidth = 2; c.gridheight=1;
    this.add(new JButton("Button #7"), c);

    c.gridx = 1; c.gridy = 5; c.gridwidth = 1; c.gridheight=1;
    this.add(new JButton("Button #8"), c);

    c.gridx = 3; c.gridy = 5; c.gridwidth = 1; c.gridheight=1;
    this.add(new JButton("Button #9"), c);
  }
}
```

Жестко запрограммированная компоновка

Все контейнеры AWT и Swing имеют принимаемый по умолчанию менеджер компоновки. Однако если установить этот менеджер в `null`, можно разместить компоненты в контейнере по своему усмотрению. Это достигается путем вызова метода `setBounds()` каждого из компонентов или, в Java 1.0, путем вызова не рекомендуемого в настоящее время к применению метода `reshape()`. Обратите внимание на то, что эти способы не сработают, если задан какой-либо менеджер компоновки, так как менеджер компоновки изменит размеры и положения всех компонентов в контейнере.

Перед применением этой техники следует хорошо уяснить себе, что по многим серьезным причинам не стоит жестко задавать размеры и положения компонентов. Во-первых, поскольку стиль компонентов может зависеть от платформы, они могут иметь различные размеры на разных платформах. Шрифты тоже несколько меняются от платформы к платформе, что может влиять на размеры компонентов. И наконец, жесткое программирование размеров и положения компонентов исключит возможность настройки (использования любимого шрифта пользователя, например) или интернационализации (перевода текста в вашем GUI на другие языки).

Тем не менее может случиться так, что управление компоновкой оказывается настолько сложным, что приходится обратиться к жесткому программированию размеров и положения компонентов. Пример 10.8 – простая программа, которая это делает; порождаемая ею компоновка показана на рис. 10.8. Обратите внимание на то, что в этом примере за-

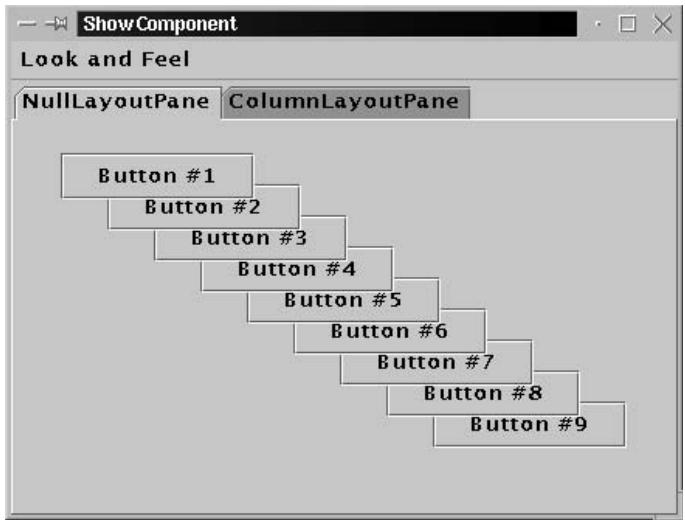


***Рис. 10.8.** Жестко запрограммированные положения компонентов*

мещается метод getPreferredSize(), запрашивающий первоначальный размер контейнера. Эта функциональная возможность обычно предоставляется менеджером компоновки, но в отсутствие менеджера приходится определять первоначальный размер контейнера самим. Поскольку используется Swing-контейнер, замещение getPreferredSize() не строго необходимо; попробуйте вместо этого вызвать setPreferredSize().

Пример 10.8. NullLayoutPane.java

```
package com.davidflanagan.examples.gui;
import java.awt.*;
import javax.swing.*;

public class NullLayoutPane extends JPanel {
  public NullLayoutPane() {
    // Освобождаемся от принимаемого по умолчанию менеджера компоновки.
    // Будем сами размещать компоненты.
    this.setLayout(null);

    // Создаем несколько кнопок и устанавливаем их размеры и положения
    // явным образом
    for(int i = 1; i <= 9; i++) {
      JButton b = new JButton("Button #" + i);
      b.setBounds(i*30, i*20, 125, 30); // в Java 1.0 используйте reshape()
      this.add(b);
    }
  }

  // Указываем, какого размера должна быть панель.
  public Dimension getPreferredSize() { return new Dimension(425, 250); }
}
```

Создание собственных менеджеров компоновки

Когда ни один из готовых менеджеров компоновки AWT не подходит для GUI, который вы хотите реализовать, вы можете написать собственный менеджер компоновки, реализовав интерфейсы LayoutManager или LayoutManager2. В действительности это легче, чем может показаться. В первую очередь, интересен метод layoutContainer(), который вызывается контейнером, когда тот хочет разместить компоненты, которые он содержит. Этот метод должен в цикле обойти все компоненты контейнера и для каждого, с применением метода setBounds(), установить его размер и положение. layoutContainer() может для каждого компонента вызвать preferredSize(), чтобы выяснить его предпочтительный размер.

Другим важным методом является preferredLayoutSize(). Этот метод должен возвращать предпочтительный размер контейнера. Обычно это тот размер, который требуется, чтобы все компоненты в нем имели свои предпочтительные размеры. Метод minimumLayoutSize() похож на предыдущий тем, что он должен возвращать минимальный допусти-

мый размер контейнера. Наконец, если ваш менеджер компоновки интересуют ограничители, задаваемые при вызове метода `add()` для размещения компонента в контейнере, можно определить метод `addLayoutComponent()`.

В примере 10.9 показан листинг программы *ColumnLayout.java*, реализации интерфейса `LayoutManager2`, которая располагает компоненты в столбце. `ColumnLayout` отличается от `BoxLayout` тем, что она позволяет задать горизонтальное выравнивание для компонентов в столбце. Пример 10.10 – это простая программа, которая использует `ColumnLayout` для создания вывода, показанного на рис. 10.9.

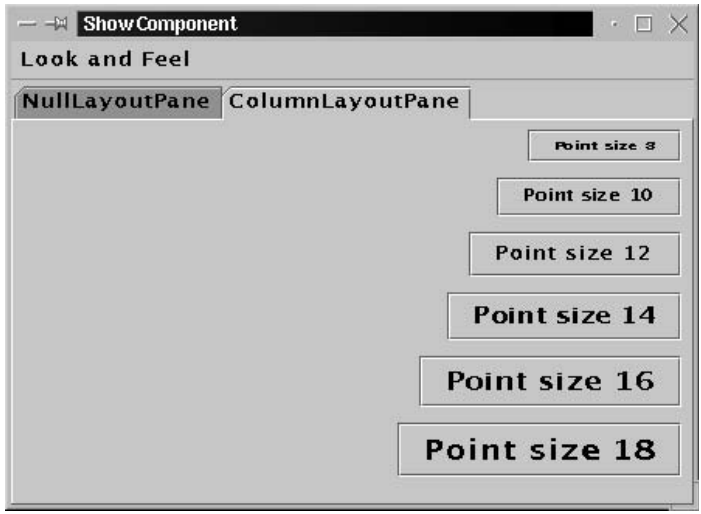


Рис. 10.9. *Компоненты, скомпонованные пользовательским менеджером компоновки*

Пример 10.9. ColumnLayout.java

```
package com.davidflanagan.examples.gui;
import java.awt.*;

/**
 * Этот LayoutManager располагает компоненты в столбце.
 * Компонентам всегда придается желаемый размер.
 *
 * При создании ColumnLayout можно задать четыре значения:
 *  margin_height - какой промежуток оставить сверху и снизу
 *  margin_width -- какой промежуток оставить справа и слева
 *  spacing -- какой промежуток по вертикали оставлять между компонентами
 *  alignment - положение компонентов по горизонтали:
 *   ColumnLayout.LEFT - компоненты выравниваются влево
 *   ColumnLayout.CENTER -- компоненты выравниваются по горизонтали по центру
```

```
 *   ColumnLayout.RIGHT -- компоненты выравниваются вправо
 *
 * Методы объекта ColumnLayout вам не придется вызывать. Следует создать
 * объект и назначить его менеджером компоновки для вашего контейнера,
 * передав его методу addLayout() объекта Container.
 */
public class ColumnLayout implements LayoutManager2 {
  protected int margin_height;
  protected int margin_width;
  protected int spacing;
  protected int alignment;

  // Константы для аргумента конструктора, задающего выравнивание.
  public static final int LEFT = 0;
  public static final int CENTER = 1;
  public static final int RIGHT = 2;

  /** Конструктор. См. выше комментарий по поводу значений его аргументов */
  public ColumnLayout(int margin_height, int margin_width,
            int spacing, int alignment) {
    this.margin_height = margin_height;
    this.margin_width = margin_width;
    this.spacing = spacing;
    this.alignment = alignment;
  }

  /**
   * Конструктор по умолчанию, создающий ColumnLayout с полями в 5 пикселов
   * в ширину и в высоту, с промежутками в 5 пикселов и выравниванием влево
   **/
  public ColumnLayout() { this(5, 5, 5, LEFT); }

  /**
   * Метод, фактически осуществляющий компоновку, вызывается контейнером
   **/
  public void layoutContainer(Container parent) {
    Insets insets = parent.getInsets();
    Dimension parent_size = parent.getSize();
    Component kid;
    int nkids = parent.getComponentCount();
    int x0 = insets.left + margin_width; // Базовая позиция по оси X
    int x;
    int y = insets.top + margin_height;  // Начинаем с верха столбца

    for(int i = 0; i < nkids; i++) {     // Цикл по компонентам
      kid = parent.getComponent(i);      // Берем компонент
      if (!kid.isVisible()) continue;    // Пропускаем скрытые компоненты
      Dimension pref = kid.getPreferredSize(); // Какого он размера?
      switch(alignment) {        // Вычисляем координату X
      default:
      case LEFT:  x = x0; break;
      case CENTER: x = (parent_size.width - pref.width)/2; break;
      case RIGHT:
```

```
        x = parent_size.width-insets.right-margin_width-pref.width;
        break;
      }
      // Устанавливаем размер и положение этого компонента
      kid.setBounds(x, y, pref.width, pref.height);
      y += pref.height + spacing;    // Получаем координату Y положения
                                     // следующего компонента
    }
  }

  /** Контейнер вызывает этот метод для определения размера,
      который должен иметь столбец (layout)*/
  public Dimension preferredLayoutSize(Container parent) {
    return layoutSize(parent, 1);
  }
  /** Контейнер вызывает этот метод для определения минимального
      допустимого размера столбца */
  public Dimension minimumLayoutSize(Container parent) {
    return layoutSize(parent, 2);
  }
  /** Контейнер вызывает этот метод для определения максимального
      допустимого размера столбца */
  public Dimension maximumLayoutSize(Container parent) {
    return layoutSize(parent, 3);
  }

  // Вычисляем минимальный, максимальный или предпочтительный размер
  // всех видимых компонентов
  protected Dimension layoutSize(Container parent, int sizetype) {
    int nkids = parent.getComponentCount();
    Dimension size = new Dimension(0,0);
    Insets insets = parent.getInsets();
    int num_visible_kids = 0;

    // Вычисляем максимальную ширину и суммарный размер всех
    // дочерних элементов (kid)
    for(int i = 0; i < nkids; i++) {
      Component kid = parent.getComponent(i);
      Dimension d;
      if (!kid.isVisible()) continue;
      num_visible_kids++;
      if (sizetype == 1) d = kid.getPreferredSize();
      else if (sizetype == 2) d = kid.getMinimumSize();
      else d = kid.getMaximumSize();
      if (d.width > size.width) size.width = d.width;
      size.height += d.height;
    }

    // Добавляем поля и прочее
    size.width += insets.left + insets.right + 2*margin_width;
    size.height += insets.top + insets.bottom + 2*margin_height;
    if (num_visible_kids > 1)
      size.height += (num_visible_kids - 1) * spacing;
```

```
    return size;
  }

  // Другие методы интерфейса LayoutManager(2), не используемые этим классом
  public void addLayoutComponent(String constraint, Component comp) {}
  public void addLayoutComponent(Component comp, Object constraint) {}
  public void removeLayoutComponent(Component comp) {}
  public void invalidateLayout(Container parent) {}
  public float getLayoutAlignmentX(Container parent) { return 0.5f; }
  public float getLayoutAlignmentY(Container parent) { return 0.5f; }
}
```

Пример 10.10. ColumnLayoutPane.java

```
package com.davidflanagan.examples.gui;
import java.awt.*;
import javax.swing.*;

public class ColumnLayoutPane extends JPanel {
  public ColumnLayoutPane() {
    // Освобождаемся от принимаемого по умолчанию менеджера компоновки.
    // Мы будем размещать компоненты сами.
    this.setLayout(new ColumnLayout(5, 5, 10, ColumnLayout.RIGHT));

    // Создаем несколько кнопок и устанавливаем их размеры
    // и положения явным образом
    for(int i = 0; i < 6; i++) {
      int pointsize = 8 + i*2;
      JButton b = new JButton("Point size " + pointsize);
      b.setFont(new Font("helvetica", Font.BOLD, pointsize));
      this.add(b);
    }
  }
}
```

Обработка событий

В предыдущих разделах, посвященных управлению компоновкой, было множество примеров, в которых компоненты `JButton` размещались всякими интересными способами. Если вы запускали эти примеры, то могли заметить, что когда вы щелкаете на кнопках, ничего интересного не происходит. Четвертый шаг при создании GUI состоит в обеспечении обработки событий, которая позволит компонентам реагировать на действия пользователя. Начиная с Java 1.1 компоненты AWT и Swing используют API обработки событий, определенный моделью компонентов JavaBeans. До Java 1.1 AWT использовал другой API, не рассматриваемый в этой главе. Мы увидим несколько примеров обработки событий с применением старой модели при изучении апплетов (см. главу 15 «Апплеты»), где эта модель еще иногда применяется в целях обратной совместимости со старыми броузерами.

В Java 1.1 и выше API обработки событий основывается на событиях и слушателях событий (event listeners). Как и все прочее в Java, события являются объектами. Объект события является экземпляром класса, производного от класса `java.util.EventObject`. Пакет `java.awt.event` определяет множество классов событий, обычно используемых компонентами AWT и Swing. Пакет `javax.swing.event` определяет дополнительные события, используемые компонентами Swing, но не компонентами AWT. И наконец, пакет `java.beans` определяет пару классов событий JavaBeans, также используемых компонентами Swing. Классы событий обычно определяют методы (или поля), предоставляющие информацию о произошедшем событии. Например, класс `java.awt.event.MouseEvent` определяет метод `getX()`, возвращающий координату X положения мыши в момент наступления события.

`EventListener` – это объект, заинтересованный в том, чтобы его уведомили о наступлении события определенного типа. Объект, генерирующий события (*источник событий*, такой как компонент `JButton`), поддерживает список слушателей и предоставляет методы, которые позволяют добавлять слушатели в этот список и удалять их из него. Когда происходит событие определенного типа, источник события уведомляет об этом все зарегистрированные слушатели. Чтобы оповестить слушатели, он сначала создает объект события, описывающий событие, а затем передает этот объект в один из методов, определенных слушателями. Конкретный вызываемый метод зависит от типа события; разные типы слушателей определяют разные методы.

Все типы слушателей являются интерфейсами, производными от интерфейса `java.util.EventListener`. Это интерфейс-метка; он не определяет никаких собственных методов[1], но существует, поэтому вы можете применить оператор `instanceof`, чтобы отличить слушатели от объектов других типов. Пакеты `java.awt.event`, `javax.swing.event` и `java.beans` определяют много интерфейсов для слушателей событий, производных от общего интерфейса `EventListener`. Каждый интерфейс слушателя предназначен для определенного типа событий и определяет один или несколько методов, которые вызываются в определенных ситуациях (например, `java.awt.MouseListener`). У методов слушателей есть одна общая черта – все они принимают объект события в качестве своего единственного аргумента.

Обратите внимание, что пакеты `java.awt.event` и `javax.swing.event` определяют только *интерфейсы* слушателей, но не *классы* слушателей. GUI должен определять собственную реализацию слушателя, реализацию, предоставляющую действительный код Java, исполняемый в ответ на событие. В этих пакетах все же определяются некие классы Java – классы адаптеров событий. *Адаптер событий* (*event adapter*) – это

[1] Этот интерфейс без методов служит просто для указания принадлежности объекта некоторому типу. – *Примеч. науч. ред.*

реализация интерфейса слушателя, состоящая исключительно из пустых методов. Многие интерфейсы слушателей определяют несколько методов, но часто мы заинтересованы в каждый определенный момент лишь в одном из этих методов. В таких случаях легче создать подкласс адаптера, заместив интересующий нас метод, а не реализовать интерфейс напрямую и быть обязанными определять все его методы.

Каждое AWT- и Swing-приложение имеет автоматически генерируемый поток, называемый *потоком диспетчеризации событий* (*event dispatch thread*) и вызывающий методы слушателей. Когда приложение запускается, главный поток строит GUI, а затем, как правило, прекращается. С этого момента все, что происходит, имеет место в потоке дипетчеризации событий, в ответ на вызов методов слушателей. Один важный побочный эффект этого состоит в том, что компоненты AWT и Swing, по соображениям повышения производительности, не являются потокобезопасными. Так, при написании многопоточной программы необходимо позаботиться о том, чтобы методы компонентов вызывались только из потока диспетчеризации событий. Вызовы методов компонентов из других потоков могут привести к случайным и трудным для обнаружения ошибкам. При необходимости создания анимации или выполнения циклически повторяемых задач с применением Swing используйте `javax.swing.Timer` вместо отдельных потоков. Если еще одному потоку действительно необходимо взаимодействовать с компонентом Swing, применяйте методы `java.awt.EventQueue invokeLater()` и `invokeAndWait()`.

Глава 2 книги «Java Foundation Classes in a Nutshell» описывает API обработки событий более подробно и также содержит таблицы, где перечислены различные типы слушателей AWT и Swing, их методы и компоненты, использующие их интерфейсы. В следующих разделах содержатся примеры, иллюстрирующие различные способы применения API обработки событий в Java, также как и API для обработки определенных типов событий ввода, использующий низкоуровневый API.

Обработка событий мыши

Пример 10.11 – это листинг *ScribblePane1.java*, простого подкласса `JPanel`, который реализует интерфейсы `MouseListener` и `MouseMotionListener`, чтобы принимать события «щелчок мышью» и «перемещение мыши». Он отвечает на эти события, проводя линии и позволяя пользователю рисовать или писать на панели. На рис. 10.10 показан пример `ScribblePane1`, запущенный внутри программы `ShowComponent`, разработанной в начале этой главы, и образчик его каллиграфических возможностей.

Обратите внимание на то, что класс `ScribblePane1` одновременно является источником событий (они генерируются родительским классом `java.awt.Component`) и слушателем. Это лучше всего заметно в конструкторе, где компонент передает себя собственным методам `addMouseListe-`

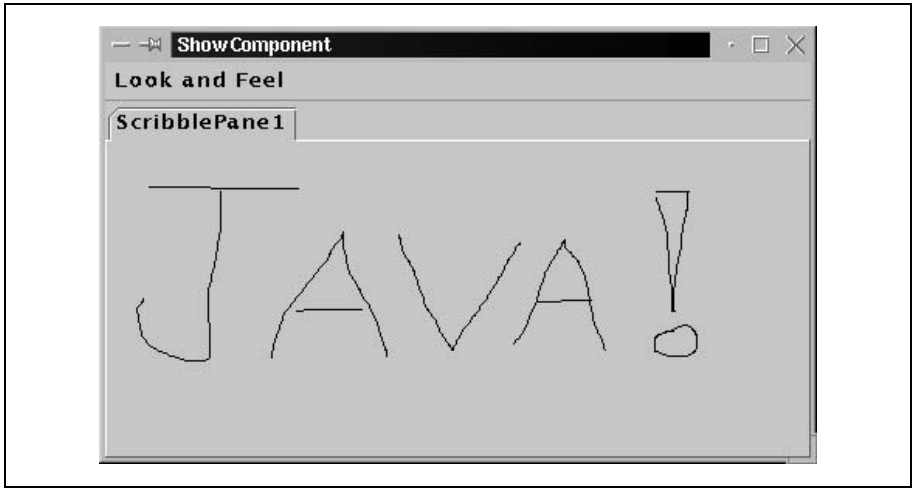


Рис. 10.10. *Надпись мышью на панели ScribblePane1*

ner() и addMouseMotionListener(). **Метод** mouseDragged() **– ключевой в рисовании: он использует метод** drawLine() **объекта** Graphics **для прочерчивания линии от предыдущего положения мыши до текущего.**

Пример 10.11. ScribblePane1.java

```
package com.davidflanagan.examples.gui;
import javax.swing.*;      // Для компонента JPanel
import java.awt.*;         // Для объекта Graphics
import java.awt.event.*;   // Для объектов Event и Listener

/**
 * Простой подкласс JPanel, использующий слушатели событий, чтобы позволить
 * пользователю рисовать мышью. Обратите внимание на то, что эти рисунки
 * не сохраняются и не перерисовываются при перерисовке окна.
 **/
public class ScribblePane1 extends JPanel
  implements MouseListener, MouseMotionListener {
  protected int last_x, last_y; // Предыдущие координаты мыши

  public ScribblePane1() {
    // Этот компонент регистрирует себя в качестве слушателя
    // событий мыши и событий перемещения мыши.
    this.addMouseListener(this);
    this.addMouseMotionListener(this);

    // Задаем предпочтительный размер компонента
    setPreferredSize(new Dimension(450,200));
  }

  // Метод из интерфейса MouseListener. Вызывается,
  // когда пользователь нажимает кнопку мыши.
  public void mousePressed(MouseEvent e) {
```

```
    last_x = e.getX(); // запоминаем координаты нажатия кнопки
    last_y = e.getY();
  }

  // Метод из интерфейса MouseMotionListener. Вызывается,
  // когда пользователь перемещает мышь при нажатой кнопке.
  public void mouseDragged(MouseEvent e) {
    int x = e.getX();  // Получаем текущее положение мыши
    int y = e.getY();
    // Проводим линию от сохраненных координат до текущей позиции
    this.getGraphics().drawLine(last_x, last_y, x, y);
    last_x = x;        // Запоминаем текущую позицию
    last_y = y;
  }

  // Прочие, неиспользуемые методы интерфейса MouseListener.
  public void mouseReleased(MouseEvent e) {}
  public void mouseClicked(MouseEvent e) {}
  public void mouseEntered(MouseEvent e) {}
  public void mouseExited(MouseEvent e) {}

  // Прочие, неиспользуемые методы интерфейса MouseMotionListener.
  public void mouseMoved(MouseEvent e) {}
}
```

Другие события мыши

Пример 10.12 показывает листинг *ScribblePane2.java*, очень похожий на `ScribblePane1`, за исключением того, что в нем в качестве слушателей используются безымянные внутренние классы. Это стандартный прием Java, используемый при программировании GUI, и фактически именно эта возможность его использования была причиной введения в язык безымянных внутренних классов. Обратите внимание на то, что внутренние классы производятся от классов адаптеров событий, а не прямо реализуют интерфейсы слушателей; а это значит, что реализовывать неиспользуемые методы необязательно.

`cribblePane2` содержит также класс `KeyListener`, который очищает панель, когда пользователь нажимает клавишу C, и определяет свойство `color` (при помощи своих методов доступа `setColor()` и `getColor()`), задающее цвет линий на рисунке. Напомню, что программа `ShowComponent` позволяет задавать значения свойств. Она понимает шестнадцатеричный RGB-формат цветов, так что при помощи следующей команды можно будет рисовать синим:

```
% java com.davidflanagan.examples.gui.ShowComponent \
 com.davidflanagan.examples.gui.ScribblePane2 color=#0000ff
```

Последнее, что нужно заметить в связи с этим примером, это то, что функция рисования была очищена и разнесена по методам `moveto()`, `lineto()` и `clear()`. Это позволяет другим компонентам вызывать дан-

ные методы и обеспечивает возможность производить подклассы от этого компонента более аккуратно.

Пример 10.12. ScribblePane2.java

```
package com.davidflanagan.examples.gui;
import javax.swing.*;     // Для компонента JPanel
import java.awt.*;        // Для объекта Graphics
import java.awt.event.*;  // Для объектов Event и Listener

/**
 * Простой подкласс JPanel, который использует слушатели событий,
 * чтобы позволить пользователю рисовать при помощи мыши. Обратите внимание
 * на то, что рисунки не сохраняются и не перерисовываются.
 **/
public class ScribblePane2 extends JPanel {
  public ScribblePane2() {
    // Задаем желательный размер компонента
    setPreferredSize(new Dimension(450,200));

    // Регистрируем обработчик событий мыши, определенный как внутренний класс
    // Обратите внимание на вызов requestFocus(). Это необходимо для того,
    // чтобы компонент принимал события клавиатуры.
    addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent e) {
          moveto(e.getX(), e.getY()); // Перемещаемся в положение щелчка
          requestFocus();             //Запрашиваем фокус ввода
        }
      });

    // Регистрируем обработчик событий движения мыши, определенный
    // как внутренний класс посредством создания подкласса MouseMotionAdapter,
    // а не реализации MouseMotionListener. Мы замещаем только интересующие
    // нас методы и наследуем принимаемые по умолчанию (пустые)
    // реализации других методов.
    addMouseMotionListener(new MouseMotionAdapter() {
        public void mouseDragged(MouseEvent e) {
          lineto(e.getX(), e.getY()); // Проводим линию к положению мыши
        }
      });

    // Добавляем обработчик событий клавиатуры, чтобы по нажатию
    // клавиши 'C' стереть рисунок
    addKeyListener(new KeyAdapter() {
        public void keyPressed(KeyEvent e) {
          if (e.getKeyCode() == KeyEvent.VK_C) clear();
        }
      });
  }

  /** Это координаты предыдущего положения мыши */
  protected int last_x, last_y;

  /** Запоминаем заданную точку */
  public void moveto(int x, int y) {
```

```
    last_x = x;
    last_y = y;
  }

  /** Проводим линию от предыдущей точки к текущей,
      затем запоминаем новую точку */
  public void lineto(int x, int y) {
    Graphics g = getGraphics(); // Получаем объект, при помощи
                                // которого будем рисовать
    g.setColor(color);          // Сообщаем ему цвет, которым надо рисовать
    g.drawLine(last_x, last_y, x, y);  // Сообщаем ему, что именно
                                       // следует рисовать
    moveto(x, y);                      // Сохраняем текущую точку
  }

  /**
   * Очищаем область рисования с использованием цвета фона компонента.
   * Этот метод требует перерисовать компонент. Поскольку у этого компонента
   * нет метода paintComponent(), ничего рисоваться не будет.
   * Тем не менее другие части компонента, такие как рамки
   * или подкомпоненты, будут нарисованы верно.
   **/
  public void clear() { repaint(); }

  /** Это поле содержит текущее значение свойства color, т. е. цвета линии*/
  Color color = Color.black;
  /** Это метод установки для свойства color */
  public void setColor(Color color) { this.color = color; }
  /** Это метод опроса для свойства color */
  public Color getColor() { return color; }

}
```

Обработка событий компонента

Два предыдущих примера показывают, как обрабатываются события клавиатуры и мыши. Это низкоуровневые события ввода, генерируемые системой и докладываемые слушателям посредством кодов из класса `java.awt.Component`. Обычно при построении GUI нет необходимости самостоятельно обрабатывать эти низкоуровневые события; вместо этого для обработки элементарных событий ввода и генерирования на их основе высокоуровневых, семантических событий используются готовые компоненты. Например, если компонент `JButton` замечает низкоуровневые нажатие и отпускание левой кнопки мыши, он генерирует высокоуровневое событие `java.awt.event.ActionEvent`, уведомляющее заинтересованные слушатели, что пользователь нажал кнопку, чтобы активизировать ее. Аналогично компонент `JList` генерирует событие `javax.swing.event.ListSelectionEvent`, когда пользователь делает свой выбор в списке.

Пример 10.13 – это листинг *ScribblePane3.java*. Этот пример произведен от `ScribblePane2` и добавляет к пользовательскому интерфейсу ком-

поненты JButton и JList. Пользователь может выбрать в списке цвет проводимых линий и, нажав на кнопку, очистить экран. Новые компоненты можно видеть на рис. 10.11. Этот пример демонстрирует реализацию интерфейсов ActionListener и ListSelectionListener для обеспечения реагирования на события, генерируемые компонентами Swing.

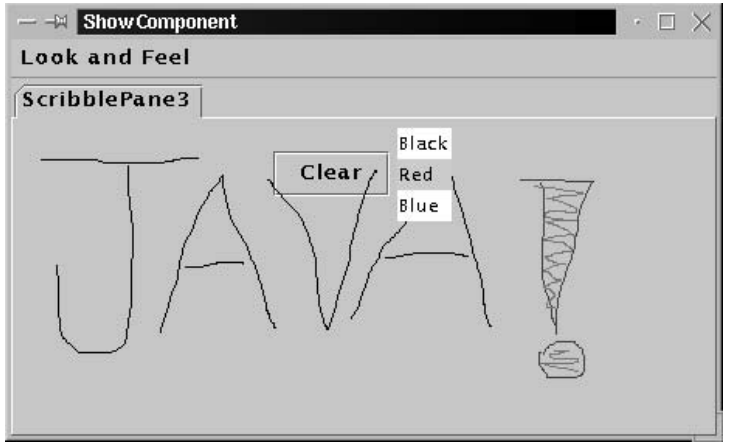


Рис. 10.11. *Компоненты Swing на панели ScribblePane3*

Запустив программу примера 10.13, обратите внимание на то, что она позволяет рисовать поверх компонентов JButton и JList. Компоненты Swing – это легковесные компоненты в том смысле, что они рисуются прямо в окне содержащего их компонента. Это совершенно отличается от случая тяжеловесных компонентов из AWT, которые используют, хотя и вложенные, но независимые окна.

Пример 10.13. ScribblePane3.java

```
package com.davidflanagan.examples.gui;
import java.awt.*;          // Для объекта Graphics и цветов
import javax.swing.*;       // Для компонента JPanel
import java.awt.event.*;    // Для интерфейса ActionListener
import javax.swing.event.*; // Для интерфейса ListSelectionListener

/**
 * Эта «грифельная доска» содержит кнопку JButton, очищающую экран,
 * и список Jlist, позволяющий пользователю выбирать цвет линий. Она
 * применяет объекты слушателей для обработки событий, генерируемых
 * этими подкомпонентами.
 **/
public class ScribblePane3 extends ScribblePane2 {
  // Это набор цветов, из которого может выбирать пользователь.
  Color[] colors = new Color[] { Color.black, Color.red, Color.blue };
  // Это имена для этих цветов
```

```
    String[] colorNames = new String[] { "Black", "Red", "Blue" };

    // Добавляем в панель компоненты JButton и JList.
    public ScribblePane3() {
      // Неявно присутствующий здесь вызов super() влечет за собой
      // вызов конструктора родительского класса

      // Помещаем в панель кнопку «Clear».
      // Обрабатываем события кнопки при помощи ActionListener
      JButton clear = new JButton("Clear");
      clear.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) { clear(); }
        });
      this.add(clear);

      // Добавляем список JList, обеспечивая выбор цвета. Обрабатываем при помощи
      // ListSelectionListener события выбора из списка.
      final JList colorList = new JList(colorNames);
      colorList.addListSelectionListener(new ListSelectionListener() {
          public void valueChanged(ListSelectionEvent e) {
            setColor(colors[colorList.getSelectedIndex()]);
          }
        });
      this.add(colorList);
    }
  }
```

Обработка событий низкого уровня

Как мы уже говорили, графические интерфейсы пользователя обычно сами не занимаются обработкой низкоуровневых событий от мыши и клавиатуры. Вместо этого они пользуются интерпретацией этих событий, которую им выдают предопределенные компоненты. Кстати говоря, предопределенные компоненты, которые обрабатывают часто происходящие события мыши и клавиатуры, обычно не используют высокоуровневые API слушателей событий.

Пример 10.14 показывает листинг *ScribblePane4.java*, окончательный вариант нашей грифельной доски. В этой версии вообще не используются слушатели, а вместо этого замещаются методы `processMouseEvent()`, `processMouseMotionEvent()` и `processKeyEvent()` базового класса `java.awt.Component`.[1] Объекты событий передаются этим методам напрямую, без всякой необходимости регистрировать слушатели. Но при этом требуется, чтобы конструктор вызвал метод `enableEvents()` для указания того, какого рода событиями он интересуется. Если конструктор этого не сделает, система, скорее всего, не доставит события этих типов, и разные методы обработки событий, возможно, никогда

[1] Точно такая же техника может применяться к другим методам класса `Component` — `processFocusEvent()`, `processComponentEvent()` и `processWindowEvent`.

не будут вызваны. Обратите внимание на то, что методы обработки событий вызывают реализацию родительского класса для всех событий, которые не обрабатывают сами. Это позволяет родительскому классу производить рассылку этих событий всем зарегистрированным слушателям.

Пример 10.14. ScribblePane4.java

```
package com.davidflanagan.examples.gui;
import javax.swing.*;     // Для компонента JPanel
import java.awt.*;        // Для объекта Graphics
import java.awt.event.*;  // Для объектов Event и Listener

/**
 * Еще один класс для рисования мышью. В этом случае вместо регистрации
 * слушателей замещаются методы обработки событий низкого уровня,
 * принадлежащие компоненту.
 **/
public class ScribblePane4 extends JPanel {
  public ScribblePane4() {
    // Задаем желательный размер компонента
    setPreferredSize(new Dimension(450,200));

    // Сообщаем системе, какого рода событиями интересуется компонент
    enableEvents(AWTEvent.MOUSE_EVENT_MASK |
           AWTEvent.MOUSE_MOTION_EVENT_MASK |
           AWTEvent.KEY_EVENT_MASK);
  }

  public void processMouseEvent(MouseEvent e) {
    if (e.getID() == MouseEvent.MOUSE_PRESSED) {
      moveto(e.getX(), e.getY());
      requestFocus();
    }
    else super.processMouseEvent(e); // Передаем необработанные события
                                     // родительскому классу
  }

  public void processMouseMotionEvent(MouseEvent e) {
    if (e.getID() == MouseEvent.MOUSE_DRAGGED) lineto(e.getX(), e.getY());
    else super.processMouseMotionEvent(e);
  }

  public void processKeyEvent(KeyEvent e) {
    if ((e.getID() == KeyEvent.KEY_PRESSED) &&
      (e.getKeyCode() == KeyEvent.VK_C)) clear();
    else super.processKeyEvent(e);  // Даем родительскому классу шанс
                                    // обработать это событие
  }

  /** Это координаты предыдущего положения мыши */
  protected int last_x, last_y;

  /** Запоминаем заданную точку */
```

```
  public void moveto(int x, int y) {
    last_x = x;
    last_y = y;
  }

  /** Проводим линию от предыдущей точки к текущей, затем запоминаем
      эту новую точку */
  public void lineto(int x, int y) {
    getGraphics().drawLine(last_x, last_y, x, y);
    moveto(x, y);
  }

  /** Очищаем область рисунка с использованием фонового цвета компонента */
  public void clear() { repaint(); }
}
```

Создание собственных событий и слушателей

Хотя Swing и AWT определяют не так много классов событий и интерфейсов слушателей, нет никаких причин, по которым мы не могли бы определить собственные типы событий и слушателей. Класс, показанный в примере 10.15, делает именно это: с применением внутренних классов он определяет собственные типы событий и слушателей.

Набор компонентов Swing дает пользователю несколько возможных способов выбрать элемент в списке. Такой выбор можно представить при помощи компонента `JList`, компонета `JComboBox` или при помощи группы кнопок `JRadioButton`. API, используемые для создания этих компонентов, манипулирования ими и для ответа на события, связанные с этими компонентами, существенно различаются. Пример 10.15 представляет собой листинг класса `ItemChooser`, который абстрагируется от разницы между этими тремя представлениями. При создании компонента `ItemChooser` задаются имя представляемого выбора, список элементов, в котором производится выбор, элемент, выбранный в настоящий момент, и тип представления. Тип представления определяет способ, каким выбор представляется пользователю, но API, используемый для работы с `ItemChooser`, не зависит от представления.

Класс `ItemChooser` включает в себя внутренний класс `Demo`. `ItemChooser.Demo` обладает методом `main()`, демонстрирующим компонент так, как это можно видеть на рис. 10.12. Демонстрационная программа получает текст вариантов выбора из командной строки, поэтому ее можно запустить, например, так:

```
% java com.davidflanagan.examples.gui.ItemChooser\$Demo Fourscore and twenty \
 years ago
```

Обратите внимание на то, что в Unix-системах следует предварить (escape) символ `$` во внутреннем классе обратным слэшем (`\`). В системах Windows это необязательно. Мы встретимся с еще одним применением `ItemChooser` в примере 10.18.

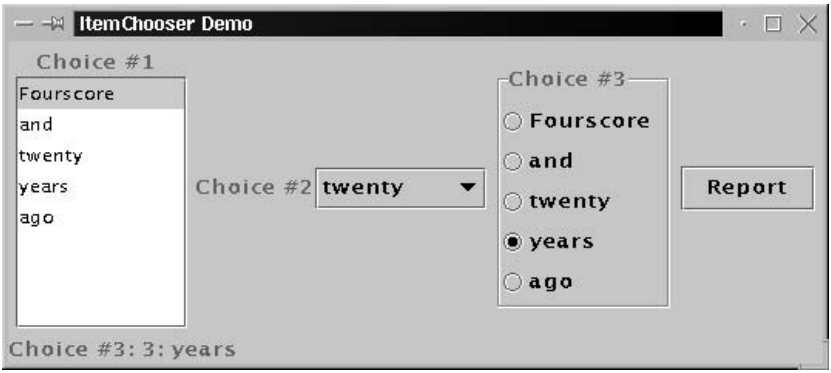


Рис. 10.12. *Компонент ItemChooser*

В компоненте `ItemChooser` особенно интересно то, что он определяет типы и своих событий, и слушателей в виде внутренних классов. Определения этих типов заслуживают внимания, равно как и их применение в `Demo`-классах `ItemChooser`, поскольку этот пример демонстрирует обе стороны архитектуры событий: генерирование событий и обработку событий. Этот пример показывает, как работать с компонентами `JList`, `JComboBox` и `JRadioButton`; особенно он интересен тем, что ожидает события, генерируемые этими внутренними компонентами, отвечает на них и приводит эти внутренние события к собственному типу событий. Разобравшись в работе `ItemChooser`, вы обретете глубокое знание архитектуры событий AWT и Swing.

Пример 10.15. ItemChooser.java

```
package com.davidflanagan.examples.gui;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;
import java.util.*;

/**
 * Этот класс представляет собой компонент Swing, предоставляющий
 * пользователю возможность выбора. Он допускает представление выбора в виде
 * компонентов JList, JComboBox или в виде заключенной в рамку группы
 * компонентов JRadioButton. К тому же он отображает имя варианта выбора
 * при помощи JLabel. Он позволяет связать с каждым возможным выбором
 * произвольное значение. Обратите внимание на то, что этот компонент
 * допускает выбор в каждый момент только одного элемента.
 * Множественный выбор не поддерживается.
 **/
public class ItemChooser extends JPanel {
```

```
// Эти поля содержат значения свойств для данного компонента
String name;         // Название списка вариантов
String[] labels;     // Текст для вариантов выбора
Object[] values;     // Произвольные значения, ассоциированные
                     // с каждым вариантом
int selection;       // Сделанный выбор
int presentation;    // Способ представления выбора

// Это допустимые значения поля presentation
public static final int LIST = 1;
public static final int COMBOBOX = 2;
public static final int RADIOBUTTONS = 3;

// Эти компоненты используются для трех допустимых типов представления
JList list;                    // Один тип представления
JComboBox combobox;            // Другой тип представления
JRadioButton[] radiobuttons;   // Еще один тип представления

// Список объектов, интересующихся нашим состоянием
ArrayList listeners = new ArrayList();

// Конструктор, устанавливающий все эти значения
public ItemChooser(String name, String[] labels, Object[] values,
          int defaultSelection, int presentation)
{
  // Копируем аргументы конструктора в поля объекта
  this.name = name;
  this.labels = labels;
  this.values = values;
  this.selection = defaultSelection;
  this.presentation = presentation;

  // Если значения не предоставлены, принимаем в качестве значений метки
  if (values == null) this.values = labels;

  // Теперь создаем обработчики событий и содержимого,
  // в зависимости от типа представления
  switch(presentation) {
  case LIST: initList(); break;
  case COMBOBOX: initComboBox(); break;
  case RADIOBUTTONS: initRadioButtons(); break;
  }
}

// Инициализация для представления JList
void initList() {
  list = new JList(labels);          // Создаем список
  list.setSelectedIndex(selection); // Устанавливаем начальное состояние

  // Обрабатываем изменения состояния
  list.addListSelectionListener(new ListSelectionListener() {
      public void valueChanged(ListSelectionEvent e) {
        ItemChooser.this.select(list.getSelectedIndex());
      }
```

```
    });

  // Располагаем имена меток в списке по вертикали
  this.setLayout(new BoxLayout(this, BoxLayout.Y_AXIS)); // вертикально
  this.add(new JLabel(name));       // Отображаем имя выбора
  this.add(new JScrollPane(list));  // Добавляем JList
}

// Инициализация для представления JComboBox
void initComboBox() {
  combobox = new JComboBox(labels);      // Создаем раскрывающийся список
  combobox.setSelectedIndex(selection);  // Устанавливаем начальное
                                         // состояние

  // Обрабатываем изменения состояния
  combobox.addItemListener(new ItemListener() {
      public void itemStateChanged(ItemEvent e) {
        ItemChooser.this.select(combobox.getSelectedIndex());
      }
    });

  // Располагаем раскрывающийся список и надпись горизонтально
  this.setLayout(new BoxLayout(this, BoxLayout.X_AXIS));
  this.add(new JLabel(name));
  this.add(combobox);
}

// Инициализация для представления JRadioButton
void initRadioButtons() {
  // Создаем массив переключателей
  radiobuttons = new JRadioButton[labels.length];   // Массив
  ButtonGroup radioButtonGroup = new ButtonGroup(); // Обеспечивает
                                                    // взаимное исключение
  ChangeListener listener = new ChangeListener() {  // Общий слушатель
      public void stateChanged(ChangeEvent e) {
        JRadioButton b = (JRadioButton)e.getSource();
        if (b.isSelected()) {
          // Если мы приняли это событие, произошедшее из-за того,
          // что какой-то переключатель был выбран, пробегаем в цикле
          // список переключателей, чтобы определить
          // индекс выбранного переключателя.
          for(int i = 0; i < radiobuttons.length; i++) {
            if (radiobuttons[i] == b) {
              ItemChooser.this.select(i);
              return;
            }
          }
        }
      }
    };

  // Отображаем имя выбора на рамке вокруг переключателей
  this.setBorder(new TitledBorder(new EtchedBorder(), name));
  this.setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
```

```
    // Создаем переключатели, помещаем их в группу и задаем для каждого
    // из них слушатель.
    for(int i = 0; i < labels.length; i++) {
      radiobuttons[i] = new JRadioButton(labels[i]);
      if (i == selection) radiobuttons[i].setSelected(true);
      radiobuttons[i].addChangeListener(listener);
      radioButtonGroup.add(radiobuttons[i]);
      this.add(radiobuttons[i]);
    }
  }

  // Эти простые методы доступа просто возвращают значения полей
  // Эти свойства – только для чтения. Их значения устанавливаются
  // конструктором и не могут меняться.
  public String getName() { return name; }
  public int getPresentation() { return presentation; }
  public String[] getLabels() { return labels; }
  public Object[] getValues() { return values; }

  /** Возвращаем индекс выбранного элемента */
  public int getSelectedIndex() { return selection; }

  /** Возвращаем объект, связанный с выбранным элементом*/
  public Object getSelectedValue() { return values[selection]; }

  /**
   * Устанавливаем выбранный элемент по заданному индексу.
   * Вызов этого метода изменяет экранный вывод, но не генерирует событий.
   **/
  public void setSelectedIndex(int selection) {
    switch(presentation) {
    case LIST: list.setSelectedIndex(selection); break;
    case COMBOBOX: combobox.setSelectedIndex(selection); break;
    case RADIOBUTTONS: radiobuttons[selection].setSelected(true); break;
    }
    this.selection = selection;
  }

  /**
   * Этот внутренний метод вызывается при изменении сделанного выбора.
   * Он сохраняет новый индекс сделанного выбора и активизирует события
   * для всех зарегистрированных слушателей. Все слушатели, зарегистрированные
   * в JList, JComboBox или JRadioButtons, вызывают этот метод.
   **/
  protected void select(int selection) {
    this.selection = selection;  // Сохраняем вновь выбранный индекс
    if (!listeners.isEmpty()) {  // Если зарегистрированы какие-то слушатели,
      // создаем объект события, описывающий сделанный выбор
      ItemChooser.Event e =
        new ItemChooser.Event(this, selection, values[selection]);
      // Обходим слушатели в цикле, используя Iterator
      for(Iterator i = listeners.iterator(); i.hasNext();) {
        ItemChooser.Listener l = (ItemChooser.Listener)i.next();
        l.itemChosen(e); // Уведомляем каждый слушатель о сделанном выборе
```

```
    }
  }
}

// Эти методы предназначены для регистрации и удаления слушателей
public void addItemChooserListener(ItemChooser.Listener l) {
  listeners.add(l);
}
public void removeItemChooserListener(ItemChooser.Listener l) {
  listeners.remove(l);
}

/**
 * Этот внутренний класс определяет тип события, генерируемого
 * объектами ItemChooser. Имя внутреннего класса Event,
 * так что его полным именем будет ItemChooser.Event
 **/
public static class Event extends java.util.EventObject {
  int selectedIndex;     // индекс выбранного элемента
  Object selectedValue;  // связанное с ним значение
  public Event(ItemChooser source,
         int selectedIndex, Object selectedValue) {
    super(source);
    this.selectedIndex = selectedIndex;
    this.selectedValue = selectedValue;
  }

  public ItemChooser getItemChooser() { return (ItemChooser)getSource();}
  public int getSelectedIndex() { return selectedIndex; }
  public Object getSelectedValue() { return selectedValue; }
}

/**
 * Этот внутренний интерфейс должен быть реализован любым объектом,
 * желающим, чтобы его уведомили, когда текущий выбранный элемент
 * в компоненте ItemChooser изменяется.
 **/
public interface Listener extends java.util.EventListener {
  public void itemChosen(ItemChooser.Event e);
}

/**
 * Этот внутренний класс служит для демонстрации компонента ItemChooser
 * Он использует аргументы, заданные в командной строке, как метки
 * и как значения для ItemChooser.
 **/
public static class Demo {
  public static void main(String[] args) {
   // Создаем окно и готовимся к выходу, если поступит требование закрыть окно
    final JFrame frame = new JFrame("ItemChooser Demo");
    frame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {System.exit(0);}
      });
```

```
// "Строка сообщения" для отображения результатов
final JLabel msgline = new JLabel(" ");

// Создаем панель, содержащую три компонента ItemChooser
JPanel chooserPanel = new JPanel();
final ItemChooser c1 = new ItemChooser("Choice #1", args, null, 0,
                    ItemChooser.LIST);
final ItemChooser c2 = new ItemChooser("Choice #2", args, null, 0,
                    ItemChooser.COMBOBOX);
final ItemChooser c3 = new ItemChooser("Choice #3", args, null, 0,
                    ItemChooser.RADIOBUTTONS);

// Слушатель, отображающий в строке сообщений произошедшие изменения
ItemChooser.Listener l = new ItemChooser.Listener() {
    public void itemChosen(ItemChooser.Event e) {
      msgline.setText(e.getItemChooser().getName() + ": " +
              e.getSelectedIndex() + ": " +
              e.getSelectedValue());
    }
  };
c1.addItemChooserListener(l);
c2.addItemChooserListener(l);
c3.addItemChooserListener(l);

// Вместо того чтобы отслеживать каждое изменение
// при помощи ItemChooser.Listener, приложения могут также
// просто опрашивать текущее состояние по мере надобности.
// Это кнопка, которая так и делает.
JButton report = new JButton("Report");
report.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      // Обратите внимание на использование многострочного курсивного
      // HTML-текста в диалоговом окне сообщений JOptionPane.
      String msg = "<html><i>" +
       c1.getName() + ": " + c1.getSelectedValue() + "<br>"+
       c2.getName() + ": " + c2.getSelectedValue() + "<br>"+
       c3.getName() + ": " + c3.getSelectedValue() + "</i>";
      JOptionPane.showMessageDialog(frame, msg);
    }
  });

// Помещаем в панель три объекта ItemChooser и кнопку Button
chooserPanel.add(c1);
chooserPanel.add(c2);
chooserPanel.add(c3);
chooserPanel.add(report);

// Помещаем в окно панель и строку сообщений
Container contentPane = frame.getContentPane();
contentPane.add(chooserPanel, BorderLayout.CENTER);
contentPane.add(msgline, BorderLayout.SOUTH);

// Устанавливаем размер окна и отображаем его.
```

```
            frame.pack();
            frame.show();
          }
        }
      }
```

Законченный GUI

Мы рассмотрели по отдельности компоненты, контейнеры, управление компоновкой и обработку событий, и теперь настало время собрать все это вместе, добавив еще некоторые детали, чтобы создать законченный графический интерфейс пользователя. В примере 10.16 приводится программа *Scribble.java*, простое приложение для рисования (рис. 10.13).

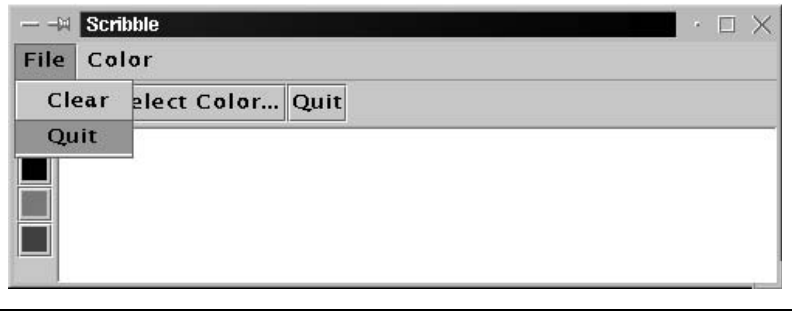


Рис. 10.13. *Приложение Scribble*

Это приложение опирается на возможности рисования, свойственные классу `ScribblePane2` из примера 10.12. Оно помещает экземпляр `ScribblePane2` в контейнер `JFrame`, главное окно приложения, а затем добавляет компонент `JMenuBar` и два компонента `JToolBar`, позволяющие пользователю управлять приложением. Приложение `Scribble` использует компонент `JColorChooser`, дающий пользователю возможность выбора цвета, и компонент `JOptionPane`, запрашивающий подтверждение запроса пользователя на выход из приложения. Следует обратить особое внимание на то, как используются эти пять компонентов Swing – большинство полноценных приложений применяют их подобным образом. Обратите внимание, что пример 10.16 – это законченное приложение; оно сконструировано для самостоятельной работы, и для его просмотра не нужна программа `ShowComponent`. Тем не менее класс `Scribble` сконструирован как подкласс `JFrame`, поэтому другие приложения, если захотят, могут создавать свои экземпляры окон `Scribble`.

Этот пример знакомит нас также с интерфейсом `Action`, производным от `ActionListener`. Всякий объект `Action` может использоваться в качестве `ActionListener` для обеспечения реакции на событие `ActionEvent`,

генерируемое компонентом. Новой в интерфейсе Action по сравнению с интерфейсом ActionListener является возможность связывать с объектом Action произвольные свойства. Интерфейс Action определяет также стандартные названия свойств, которые могут задавать имя, графический символ (значок) и описание действия, осуществляемого слушателем. Удобство применения объектов Action связано с тем, что они могут быть прямо помещены внутрь компонента JMenu и JToolBar; компоненты используют имя действия и/или его значок для автоматического создания представляющих действие элементов меню или кнопок в панели инструментов. (В Java 1.3 объекты Action могут также прямо передаваться методам-конструкторам таких компонентов, как, например, JButton.) Объекты действий могут также разрешаться и запрещаться. Когда действие запрещено, любой компонент, созданный для его представления, также запрещается, пресекая попытки пользователя выполнить это действие.

Пример 10.16. Scribble.java

```
package com.davidflanagan.examples.gui;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

/**
 * Этот подкласс JFrame является простым "приложением для рисования".
 **/
public class Scribble extends JFrame {
  /**
   * Метод main() создает экземпляр класса, устанавливает его размер
   * и отображает его на экране
   **/
  public static void main(String[] args) {
    Scribble scribble = new Scribble();
    scribble.setSize(500, 300);
    scribble.setVisible(true);
  }

  // «Приложение для рисования» использует разработанный ранее компонент
  // ScribblePane2. Это поле содержит используемый экземпляр ScribblePane2.
  ScribblePane2 scribblePane;

  /**
   * Этот конструктор создает GUI для этого приложения.
   **/
  public Scribble() {
    super("Scribble"); // Вызываем конструктор родительского класса
                       // и устанавливаем название окна

    // Обрабатываем команду закрыть окно
    this.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) { System.exit(0); }
      });
```

```
// Все содержимое окна JFrame (за исключением полосы меню) принадлежит
// внутренней "панели содержимого" окна, а не самому окну. То же
// справедливо и для JDialog и подобных контейнеров верхнего уровня.
Container contentPane = this.getContentPane();

// Задаем менеджер компоновки для панели содержимого
contentPane.setLayout(new BorderLayout());

// Создаем компонент главной панели для рисования, снабжаем его рамкой,
// задаем цвет фона и помещаем его в панель содержимого
scribblePane = new ScribblePane2();
scribblePane.setBorder(new BevelBorder(BevelBorder.LOWERED));
scribblePane.setBackground(Color.white);
contentPane.add(scribblePane, BorderLayout.CENTER);

// Создаем полосу меню и помещаем ее в окно. Обратите внимание на то,
// что JFrame обрабатывает меню особо и имеет специальный метод
// для размещения меню вне панели содержимого.
JMenuBar menubar = new JMenuBar(); // Создаем полосу меню
this.setJMenuBar(menubar);         // Отображаем ее в окне JFrame

// Создаем меню и вставляем его в полосу меню
JMenu filemenu = new JMenu("File");
JMenu colormenu = new JMenu("Color");
menubar.add(filemenu);
menubar.add(colormenu);

// Создаем несколько объектов Action для использования с меню и панелями
// инструментов. Action соединяет надпись меню и/или значок
// с объектом ActionListener. Эти классы Action ниже
// определяются в качестве внутренних классов.
Action clear = new ClearAction();
Action quit = new QuitAction();
Action black = new ColorAction(Color.black);
Action red = new ColorAction(Color.red);
Action blue = new ColorAction(Color.blue);
Action select = new SelectColorAction();

// Заполняем меню объектами Action
filemenu.add(clear);
filemenu.add(quit);
colormenu.add(black);
colormenu.add(red);
colormenu.add(blue);
colormenu.add(select);

// Теперь создаем панель инструментов, помещаем в нее действия, а ее саму
// размещаем в верхней части окна (где она появится под полосой меню)
JToolBar toolbar = new JToolBar();
toolbar.add(clear);
toolbar.add(select);
toolbar.add(quit);
contentPane.add(toolbar, BorderLayout.NORTH);
```

```
    // Создаем другую панель инструментов, которая будет использоваться
    // в качестве палитры, и размещаем ее у левого края окна.
    JToolBar palette = new JToolBar();
    palette.add(black);
    palette.add(red);
    palette.add(blue);
    palette.setOrientation(SwingConstants.VERTICAL);
    contentPane.add(palette, BorderLayout.WEST);
  }

  /** Этот внутренний класс определяет действие "Clear", стирающее рисунок */
  class ClearAction extends AbstractAction {
    public ClearAction() {
      super("Clear"); // Задаем название действия
    }
    public void actionPerformed(ActionEvent e) { scribblePane.clear(); }
  }

  /** Этот внутренний класс определяет действие "Quit", производящее выход
      из программы */
  class QuitAction extends AbstractAction {
    public QuitAction() { super("Quit"); }
    public void actionPerformed(ActionEvent e) {
      // Применяем JOptionPane для получения от пользователя
      // подтверждения намерения выйти
      int response =
        JOptionPane.showConfirmDialog(Scribble.this, "Really Quit?");
      if (response == JOptionPane.YES_OPTION) System.exit(0);
    }
  }

  /**
   * Этот внутренний класс определяет класс типа Action, устанавливающий цвет
   * рисуемых линий для компонента ScribblePane2. Обратите внимание на то,
   * что действия этого типа имеют значки, а не текстовые метки
   **/
  class ColorAction extends AbstractAction {
    Color color;
    public ColorAction(Color color) {
      this.color = color;
      putValue(Action.SMALL_ICON, new ColorIcon(color)); // задаем значки
    }
    public void actionPerformed(ActionEvent e) {
      scribblePane.setColor(color); // Устанавливаем текущий цвет линий
    }
  }

  /**
   * Этот внутренний класс реализует Icon, с целью изобразить сплошной
   * прямоугольник 16x16 заданного цвета. Большая часть значков являются
   * экземплярами ImageIcon, но поскольку здесь мы используем только сплошные
   * цвета, будет удобнее реализовать наш собственный тип Icon
   **/
```

```
  static class ColorIcon implements Icon {
    Color color;
    public ColorIcon(Color color) { this.color = color; }
    // Эти два метода задают размер значка
    public int getIconHeight() { return 16; }
    public int getIconWidth() { return 16; }
    // Этот метод рисует значок
    public void paintIcon(Component c, Graphics g, int x, int y) {
      g.setColor(color);
      g.fillRect(x, y, 16, 16);
    }
  }

  /**
   * Этот внутренний класс определяет Action с использованием JColorChooser,
   * чтобы предоставить пользователю возможность выбрать цвет линий
   **/
  class SelectColorAction extends AbstractAction {
    public SelectColorAction() { super("Select Color..."); }
    public void actionPerformed(ActionEvent e) {
      Color color = JColorChooser.showDialog(Scribble.this,
                          "Select Drawing Color",
                          scribblePane.getColor());
      if (color != null) scribblePane.setColor(color);
    }
  }
}
```

Действия и отражение

Пример 10.16 демонстрирует применение объектов `Action`, которые позволяют легко предоставить пользователю множество команд приложения при помощи меню, панелей инструментов и т. п. Неудобство работы с объектами `Action` состоит в том, что обычно каждый из них должен быть определен как класс. Если, однако, вы готовы применить отражения Java API, у вас есть более удобная альтернатива. В примере 8.2 главы 8 «Отражение» мы видели класс `Command` – класс, инкапсулирующий объект `java.lang.reflect.Method`, массив аргументов метода и объект, метод которого вызывается. Вызов метода `invoke()` объекта `Command` влечет за собой вызов этого метода. Самой сильной стороной класса `Command` при этом является его статический метод `parse()`, который умеет создавать объект `Command` путем анализа текстового представления имени метода и списка аргументов.

Класс `Command` реализует интерфейс `ActionListener`, поэтому объекты `Command` можно использовать в качестве простых слушателей действий. Но `Command` – это не объект типа `Action`. Пример 10.17 решает эту проблему; он представляет листинг *CommandAction.java* подкласса `AbstractAction`, который использует объект `Command` для выполнения дейст-

вия. Поскольку самую трудную часть работы выполняет класс `Command`**, код** `CommandAction` **относительно прост.**

Пример 10.17. CommandAction.java

```
package com.davidflanagan.examples.gui;
import com.davidflanagan.examples.reflect.*;
import javax.swing.*;
import java.awt.event.*;

public class CommandAction extends AbstractAction {
  Command command; // Команда, которая должна исполняться в ответ
                   // на ActionEvent

  /**
   * Создаем объект Action, который содержит указанные атрибуты и вызывает
   * заданный объект Command в ответ на события ActionEvent
   **/
  public CommandAction(Command command, String label,
            Icon icon, String tooltip,
            KeyStroke accelerator, int mnemonic,
            boolean enabled)
  {
    this.command = command; // Запоминаем команду, которую надо вызвать

    // Устанавливаем различные атрибуты действия при помощи метода putValue()
    if (label != null) putValue(NAME, label);
    if (icon != null) putValue(SMALL_ICON, icon);
    if (tooltip != null) putValue(SHORT_DESCRIPTION, tooltip);
    if (accelerator != null) putValue(ACCELERATOR_KEY, accelerator);
    if (mnemonic != KeyEvent.VK_UNDEFINED)
      putValue(MNEMONIC_KEY, new Integer(mnemonic));

    // Сообщаем действию, разрешено оно в настоящий момент или нет
    setEnabled(enabled);
  }

  /**
   * Этот метод реализует ActionListener, являющийся родительским интерфейсом
   * для Action. Когда компонент генерирует ActionEvent, оно передается
   * этому методу. Этот метод просто передает его объекту Command,
   * который также является объектом типа ActionListener
   **/
  public void actionPerformed(ActionEvent e) { command.actionPerformed(e); }

  // В Java 1.3 эти константы определены в Action.
  // Для совместимости с Java 1.2 мы замещаем их здесь.
  public static final String ACCELERATOR_KEY = "AcceleratorKey";
  public static final String MNEMONIC_KEY = "MnemonicKey";
}
```

Создание собственных диалоговых окон

Программа Scribble из примера 10.18 показывет диалоговые окна (диалоги) двух видов: диалоговое окно подтверждения, созданное при помощи JOptionPane, и диалоговое окно выбора цвета, созданное при помощи JColorChooser. Эти компоненты Swing поддерживают все часто используемые типы диалоговых окон. Класс JOptionPane позволяет легко отображать простые (и не только простые) информационные диалоги, диалоги подтверждения и выбора, тогда как JColorChooser и JFileChooser предоставляют возможность выбора цвета и файла. Большинство нетривиальных приложений, однако, нуждаются в создании собственных диалогов, для которых недостаточно возможностей стандартных компонентов. Создать собственный диалог нетрудно при помощи компонента JDialog.

Пример 10.18 показывает класс FontChooser. Он произведен от класса JDialog и использует класс ItemChooser, созданный в примере 10.15, для показа пользователю гарнитур (families), начертаний (styles) и размеров шрифтов. Диалог FontChooser изображен на рис. 10.14. Внутренний класс FontChooser.Demo представляет собой простое демонстрационное приложение, на котором можно поэкспериментировать с диалогом FontChooser.

***Рис. 10.14.** The FontChooser dialog*

Компонент JDialog имеет тип RootPaneContainer подобно JFrame, а это означает, что прямо помещать в него дочерние элементы нельзя. Вместо этого их помещают в контейнер, возвращаемый методом getContentPane(). FontChooser создает модальный диалог, это означает, что метод show() блокируется и не возвращает управление вызывающему методу, пока пользователь не закончит работу с диалогом. Наконец, FontChooser реализован как подкласс JDialog, поэтому он может использоваться другими приложениями. Если возникает потребность в создании диалога, специфичного для отдельного приложения, можно просто создать экземпляр JDialog и по собственному усмотрению заполнить его всевозможными дочерними компонентами; другими словами, для определения нового диалога не нужно создавать специальный подкласс.

Пример 10.18. FontChooser.java

```
package com.davidflanagan.examples.gui;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import com.davidflanagan.examples.gui.ItemChooser;

/**
 * Это подкласс JDialog, позволяющий пользователю в списке доступных
 * в системе шрифтов выбрать шрифт любого стиля и размера. Диалог является
 * модальным. Он отображается при помощи метода show(); этот метод
 * не возвращается к месту вызова, пока пользователь не закончит работу
 * с диалогом. Когда show() возвращается, вызывается метод etSelectedFont(),
 * получающий пользовательский выбор. Если пользователь щелкает по кнопке
 * диалога "Cancel", getSelectedFont() возвратит null.
 **/
public class FontChooser extends JDialog {
  // Эти поля определяют свойства компонента
  String family;        // Имя гарнитуры
  int style;            // Начертание шрифта
  int size;             // Размер шрифта
  Font selectedFont;    // Шрифт (Font), которому они соответствуют

  // Это список всех присутствующих в системе гарнитур шрифтов
  String[] fontFamilies;

  // Различные компоненты Swing, используемые в диалоге
  ItemChooser families, styles, sizes;
  JTextArea preview;
  JButton okay, cancel;

  // Названия начертаний из меню "Style"
  static final String[] styleNames = new String[] {
    "Plain", "Italic", "Bold", "BoldItalic"
  };
  // Значения, соответствующие различным названиям
  static final Integer[] styleValues = new Integer[] {
    new Integer(Font.PLAIN), new Integer(Font.ITALIC),
    new Integer(Font.BOLD), new Integer(Font.BOLD+Font.ITALIC)
  };
  // «Названия» размеров, которые появятся в меню размеров
  static final String[] sizeNames = new String[] {
     "8", "10", "12", "14", "18", "20", "24", "28", "32",
    "40", "48", "56", "64", "72"
  };

  // Это принимаемая по умолчанию строка предварительного просмотра,
  // отображаемая в окне диалога
  static final String defaultPreviewString =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ\n" +
    "abcdefghijklmnopqrstuvwxyz\n" +
    "1234567890!@#$%^&*()_-=+[]{}<,.>\n" +
```

```
    "Быстрая рыжая лиса перепрыгнет через ленивую собаку";

  /** Создаем диалог выбора шрифта в заданном окне. */
  public FontChooser(Frame owner) {
    super(owner, "Choose a Font"); // Устанавливаем диалоговое окно
                                   // и его название

    // Этот диалог должен использоваться как модальный. Чтобы использоваться
    // в качестве немодального (modeless) диалога, он должен был бы
    // активизировать событие PropertyChangeEvent всякий раз, когда меняется
    // выбор шрифта, чтобы сообщить приложению о сделанном выборе.
    setModal(true);

    // Выясняем, какие шрифты доступны в системе
    GraphicsEnvironment env =
      GraphicsEnvironment.getLocalGraphicsEnvironment();
    fontFamilies = env.getAvailableFontFamilyNames();

    // Устанавливаем начальные значения свойств
    family = fontFamilies[0];
    style = Font.PLAIN;
    size = 18;
    selectedFont = new Font(family, style, size);

    // Создаем объекты ItemChooser, позволяющие пользователю выбирать
    // гарнитуру шрифта, его начертание и размер.
    families = new ItemChooser("Family", fontFamilies, null, 0,
                  ItemChooser.COMBOBOX);
    styles = new ItemChooser("Style", styleNames, styleValues, 0,
                 ItemChooser.COMBOBOX);
    sizes = new ItemChooser("Size", sizeNames,null,4,ItemChooser.COMBOBOX);

    // Теперь регистрируем слушатели, обрабатывающие выборы
    families.addItemChooserListener(new ItemChooser.Listener() {
        public void itemChosen(ItemChooser.Event e) {
          setFontFamily((String)e.getSelectedValue());
        }
      });
    styles.addItemChooserListener(new ItemChooser.Listener() {
        public void itemChosen(ItemChooser.Event e) {
          setFontStyle(((Integer)e.getSelectedValue()).intValue());
        }
      });
    sizes.addItemChooserListener(new ItemChooser.Listener() {
        public void itemChosen(ItemChooser.Event e) {
          setFontSize(Integer.parseInt((String)e.getSelectedValue()));
        }
      });

    // Создаем компонент для просмотра шрифта.
    preview = new JTextArea(defaultPreviewString, 5, 40);
    preview.setFont(selectedFont);

   // Создаем кнопки, прекращающие диалог, и устанавливаем обработчики для них
```

```
    okay = new JButton("Okay");
    cancel = new JButton("Cancel");
    okay.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) { hide(); }
      });
    cancel.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          selectedFont = null;
          hide();
        }
      });

    // Помещаем компоненты ItemChooser в компонент Box
    Box choosersBox = Box.createHorizontalBox();
    choosersBox.add(Box.createHorizontalStrut(15));
    choosersBox.add(families);
    choosersBox.add(Box.createHorizontalStrut(15));
    choosersBox.add(styles);
    choosersBox.add(Box.createHorizontalStrut(15));
    choosersBox.add(sizes);
    choosersBox.add(Box.createHorizontalStrut(15));
    choosersBox.add(Box.createGlue());

    // Помещаем кнопки завершения диалога в другой компонент Box
    Box buttonBox = Box.createHorizontalBox();
    buttonBox.add(Box.createGlue());
    buttonBox.add(okay);
    buttonBox.add(Box.createGlue());
    buttonBox.add(cancel);
    buttonBox.add(Box.createGlue());

    // Помещаем элементы выбора наверху, кнопки внизу,
    // а окно просмотра посередине.
    Container contentPane = getContentPane();
    contentPane.add(new JScrollPane(preview), BorderLayout.CENTER);
    contentPane.add(choosersBox, BorderLayout.NORTH);
    contentPane.add(buttonBox, BorderLayout.SOUTH);

    // Устанавливаем размер диалогового окна на основе размеров компонентов.
    pack();
}

/**
 * Этот метод вызывается после show(), чтобы получить выбор пользователя.
 * Если пользователь нажал кнопку «Cancel», метод возвращает null
 **/
public Font getSelectedFont() { return selectedFont; }

// Это другие методы опроса свойств
public String getFontFamily() { return family; }
public int getFontStyle() { return style; }
public int getFontSize() { return size; }
```

```
// Методы установки несколько сложнее. Обратите внимание на то,
// что ни один из этих методов установки не обновляет соответствующие
// свойства компонентов ItemChooser, как они обычно это делают.
public void setFontFamily(String name) {
  family = name;
  changeFont();
}
public void setFontStyle(int style) {
  this.style = style;
  changeFont();
}
public void setFontSize(int size) {
  this.size = size;
  changeFont();
}
public void setSelectedFont(Font font) {
  selectedFont = font;
  family = font.getFamily();
  style = font.getStyle();
  size = font.getSize();
  preview.setFont(font);
}

// Этот метод вызывается, когда изменяются гарнитура, начертание
// или размер шрифта
protected void changeFont() {
  selectedFont = new Font(family, style, size);
  preview.setFont(selectedFont);
}

// Замещаем этот унаследованный метод, чтобы никто не сделал
// наш диалог немодальным
public boolean isModal() { return true; }

/** Этот внутренний класс демонстрирует применение класса FontChooser */
public static class Demo {
  public static void main(String[] args) {
    // Создаем несколько компонентов и диалог FontChooser
    final JFrame frame = new JFrame("demo");
    final JButton button = new JButton("Push Me!");
    final FontChooser chooser = new FontChooser(frame);

    // Обрабатываем нажатие кнопки
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          // Показываем диалог
          chooser.show();
          // Получаем выбор пользователя
          Font font = chooser.getSelectedFont();
          // Если отмены не было, устанавливаем шрифт для кнопки
          if (font != null) button.setFont(font);
        }
      });
```

```
        // Отображаем демонстрационное окно
        frame.getContentPane().add(button);
        frame.setSize(200, 100);
        frame.show();
      }
    }
}
```

Отображение таблиц

Теперь, после того как мы собрали образец Swing GUI, можно двинуться дальше и изучить более сложные темы, касающиеся Swing-программирования. Мы начнем с примеров более мощных, но и более сложных компонентов. Класс `JTable` отображает табличные данные. Он особенно легко применяется, когда данные организованы в виде массива массивов. Если же это не так, приходится реализовывать интерфейс `javax.swing.table.TableModel`, который будет служить переводчиком между данными и компонентом `JTable`.

Пример 10.19, листинг *PropertyTable.java*, как раз этим и занимается. `PropertyTable` является подклассом `Jtable`, который использует специальную реализацию интерфейса `TableModel` для отображения таблицы свойств, определенных заданным классом JavaBeans. Этот пример включает в себя метод `main()`, поэтому его можно запустить как самостоятельное приложение. Рисунок 10.15 показывает класс `PropertyTable` в действии. При изучении этого примера уделите особое внимание реализации `TableModel`; `TableModel` имеет решающее значение для работы с компонентом `JTable`. Обратите также внимание на конструктор `PropertyTable`, который использует `TableColumnModel`, чтобы изменить принимаемый по умолчанию внешний вид столбцов таблицы.

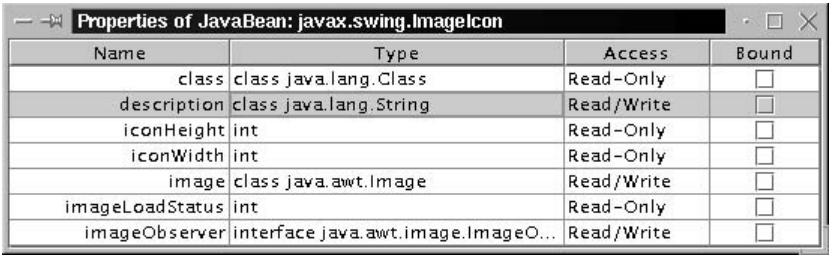


Рис. 10.15. *Приложение PropertyTable*

Пример 10.19. PropertyTable.java

```
package com.davidflanagan.examples.gui;
import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;  // TableModel и другие классы, связанные с JTable
```

```
import java.beans.*;      // Для интроспекции JavaBean
import java.util.*;       // Для сортировки массивов

/**
 * Этот класс является подклассом JTable, отображающим таблицу
 * свойств JavaBeans любого заданного класса.
 **/
public class PropertyTable extends JTable {
  /** Этот метод main() позволяет продемонстрировать класс */
  public static void main(String[] args) {
    // Задаем имя класса как аргумент командной строки
    Class beanClass = null;
    try {
      // Используем отражение для получения класса по имени класса
      beanClass = Class.forName(args[0]);
    }
    catch (Exception e) { // Сообщаем об ошибках
      System.out.println("Не найден заданный класс: "+e.getMessage());
      System.out.println("Формат: java TableDemo <JavaBean class name>");
      System.exit(0);
    }

    // Создаем таблицу для отображения свойств заданного класса
    JTable table = new PropertyTable(beanClass);

    // Затем помещаем таблицу в прокручиваемое окно, помещаем прокручиваемое
    // окно в окно верхнего уровня и выводим на экран
    JScrollPane scrollpane = new JScrollPane(table);
    JFrame frame = new JFrame("Свойства компонента JavaBean: " + args[0]);
    frame.getContentPane().add(scrollpane);
    frame.setSize(500, 400);
    frame.setVisible(true);
  }

  /**
   * Этот конструктор задает данные, которые будут отображаться в таблице
   * (модель таблицы), и использует TableColumnModel для настройки
   * способа их отображения. Самая сложная часть работы выполняется
   * определенной ниже реализацией TableModel.
   **/
  public PropertyTable(Class beanClass) {
    // Устанавливаем модель данных для этой таблицы
    try {
      setModel(new JavaBeanPropertyTableModel(beanClass));
    }
    catch (IntrospectionException e) {
      System.err.println("Предупреждение: не могу выполнить интроспекцию: " +
                         beanClass);
    }

    // Организуем макет таблицы, манипулируя ее моделью столбцов
    TableColumnModel colmodel = getColumnModel();

    // Устанавливаем ширину столбцов
```

```
    colmodel.getColumn(0).setPreferredWidth(125);
    colmodel.getColumn(1).setPreferredWidth(200);
    colmodel.getColumn(2).setPreferredWidth(75);
    colmodel.getColumn(3).setPreferredWidth(50);

    // Выравниваем текст в первом столбце по правому краю
    TableColumn namecol = colmodel.getColumn(0);
    DefaultTableCellRenderer renderer = new DefaultTableCellRenderer();
    renderer.setHorizontalAlignment(SwingConstants.RIGHT);
    namecol.setCellRenderer(renderer);
  }

  /**
   * Этот класс реализует TableModel и представляет свойства JavaBeans
   * тем способом, которым их может отобразить компонент JTable. Если нужно
   * отобразить данные табличного типа, необходимо реализовать класс TableModel,
   * чтобы описать эти данные, и компонент JTable сможет отобразить их.
   **/
  static class JavaBeanPropertyTableModel extends AbstractTableModel {
    PropertyDescriptor[] properties; // Отображаемые свойства

    /**
     * Конструктор: применяется механизм интроспекции JavaBeans
     * для получения информации о компоненте JavaBeans. Получив эту
     * информацию, другие методы смогут преобразовать ее для JTable.
     **/
    public JavaBeanPropertyTableModel(Class beanClass)
      throws java.beans.IntrospectionException
    {
      // Применяем класс-интроспектор для получения "bean info" о классе.
      BeanInfo beaninfo = Introspector.getBeanInfo(beanClass);
      // Получаем дескрипторы свойств от этого класса BeanInfo
      properties = beaninfo.getPropertyDescriptors();
      // Теперь сортируем эти свойства по имени, без учета регистра
      // Безымянная реализация Comparator задает способ сортировки
      // объектов PropertyDescriptor по имени
      Arrays.sort(properties, new Comparator() {
          public int compare(Object p, Object q) {
            PropertyDescriptor a = (PropertyDescriptor) p;
            PropertyDescriptor b = (PropertyDescriptor) q;
            return a.getName().compareToIgnoreCase(b.getName());
          }
          public boolean equals(Object o) { return o == this; }
        });
    }

    // Это имена столбцов, представленных в этом TableModel
    static final String[] columnNames = new String[] {
      "Name", "Type", "Access", "Bound"
    };

    // Это типы столбцов, представленных в этом TableModel
    static final Class[] columnTypes = new Class[] {
```

```
        String.class, Class.class, String.class, Boolean.class
      };

      // Эти простые методы возвращают основную информацию о таблице
      public int getColumnCount() { return columnNames.length; }
      public int getRowCount() { return properties.length; }
      public String getColumnName(int column) { return columnNames[column]; }
      public Class getColumnClass(int column) { return columnTypes[column]; }

      /**
       * Этот метод возвращает значение, которое выводится на пересечении
       * заданных строки и столбца таблицы
       **/
      public Object getValueAt(int row, int column) {
        PropertyDescriptor prop = properties[row];
        switch(column) {
        case 0: return prop.getName();
        case 1: return prop.getPropertyType();
        case 2: return getAccessType(prop);
        case 3: return new Boolean(prop.isBound());
        default: return null;
        }
      }

      // Вспомогательный метод, вызываемый приведенным
      // выше методом getValueAt()
      String getAccessType(PropertyDescriptor prop) {
        java.lang.reflect.Method reader = prop.getReadMethod();
        java.lang.reflect.Method writer = prop.getWriteMethod();
        if ((reader != null) && (writer != null)) return "Чтение/Запись";
        else if (reader != null) return "Только чтение";
        else if (writer != null) return "Только запись";
        else return "Нет доступа"; // Так не должно быть
      }
    }
  }
```

Отображение деревьев

Компонент `JTree` применяется для отображения данных, имеющих структуру дерева. Если данные имеют форму вложенных массивов, векторов или хеш-таблиц, можно передать корневой узел структуры данных конструктору `JTree`, и он их отобразит. Данные, имеющие древовидную структуру, обычно имеют иную форму, чем перечисленные выше. Отобразить такие данные можно, реализовав интерфейс `javax.swing.Tree.TreeModel`, чтобы проинтерпретировать данные способом, пригодным для использования компонентом `JTree`.

Пример 10.20 предъявляет листинг *ComponentTree.java* подкласса `JTree`, который использует специальную реализацию `TreeModel` для отображения иерархии вложенных контейнеров AWT или Swing GUI

в виде дерева. В классе имеется метод `main()`, который использует класс `ComponentTree` для отображения собственной иерархии компонентов, как показано на рис. 10.16. Как и в предыдущем примере с `JTable`, ядром примера является реализация интерфейса `TreeModel`. Метод `main()` иллюстрирует также приемы реагирования на события выбора вершины дерева.

Рис. 10.16. *Приложение ComponentTree*

Пример 10.20. ComponentTree.java

```
package com.davidflanagan.examples.gui;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;

/**
 * Этот класс является подклассом JTree, отображающим дерево компонентов
 * AWT или Swing, составляющих GUI.
 **/
public class ComponentTree extends JTree {
  /**
   * Все, что должен сделать этот метод-конструктор, - это подготовить
   * для дерева объекты TreeModel и TreeCellRenderer. Именно эти классы
   * (определенные ниже) делают всю настоящую работу.
   **/
  public ComponentTree(Component c) {
    super(new ComponentTreeModel(c));
    setCellRenderer(new ComponentCellRenderer(getCellRenderer()));
  }

  /**
   * Класс TreeModel придает иерархическим данным форму, которую JTree умеет
   * отображать. Эта реализация интерпретирует иерархию контейнеров компонента
   * Component для отображения при помощи класса ComponentTree. Обратите
```

```
 * внимание, что в вершинах дерева могут размещаться объекты любого типа -
 * нужно только, чтобы модель TreeModel знала, как с ними работать.
 **/
static class ComponentTreeModel implements TreeModel {
  Component root;  // Корневой объект дерева

  // Конструктор: просто запоминает корневой объект
  public ComponentTreeModel(Component root) { this.root = root; }

  // Возвращает корень дерева
  public Object getRoot() { return root; }

  // Является эта вершина концевой (листом, leaf)?
  // (Листья JTree отображает особым способом). Всякая вершина,
  // не являющаяся контейнером, является листом, поскольку у нее
  // не может быть дочерних компонентов. Пустые контейнеры также
  // определяются в качестве листьев.
  public boolean isLeaf(Object node) {
    if (!(node instanceof Container)) return true;
    Container c = (Container) node;
    return c.getComponentCount() == 0;
  }

  // Сколько дочерних элементов у этой вершины?
  public int getChildCount(Object node) {
    if (node instanceof Container) {
      Container c = (Container) node;
      return c.getComponentCount();
    }
    return 0;
  }

  // Возвращает заданный (индексом) дочерний элемент для вершины parent.
  public Object getChild(Object parent, int index) {
    if (parent instanceof Container) {
      Container c = (Container) parent;
      return c.getComponent(index);
    }
    return null;
  }

  // Возвращает индекс вершины child по отношению к вершине parent
  public int getIndexOfChild(Object parent, Object child) {
    if (!(parent instanceof Container)) return -1;
    Container c = (Container) parent;
    Component[] children = c.getComponents();
    if (children == null) return -1;
    for(int i = 0; i < children.length; i++) {
      if (children[i] == child) return i;
    }
    return -1;
  }

  // Этот метод требуется, только когда дерево редактируется, поэтому здесь
  // он не реализован.
```

```
  public void valueForPathChanged(TreePath path, Object newvalue) {}

  // TreeModel не генерирует никаких событий (поскольку дерево
  // не подлежит редактированию), поэтому методы регистрации слушателей
  // оставлены без реализации
  public void addTreeModelListener(TreeModelListener l) {}
  public void removeTreeModelListener(TreeModelListener l) {}
}

/**
 * TreeCellRenderer отображает каждую вершину дерева. Принимаемый
 * по умолчанию "отображатель" (renderer) использует для отображения
 * произвольного объекта в вершине его метод toString().
 * Метод Component.toString() возвращает длинные строки с избыточной
 * информацией, поэтому мы применяем эту промежуточную реализацию
 * TreeCellRenderer для преобразования вершин из объектов Component
 * в подходящие строковые значения перед тем, как передать эти строковые
 * значения принимаемому по умолчанию "отображателю".
 **/
static class ComponentCellRenderer implements TreeCellRenderer {
  TreeCellRenderer renderer; // "Отображатель", для которого
                             // подготавливаются данные
  // Конструктор: просто запоминает "отображатель"
  public ComponentCellRenderer(TreeCellRenderer renderer) {
    this.renderer = renderer;
  }

  // Это единственный метод TreeCellRenderer.
  // Вычисляется отображаемая строка и передается "отображателю"
  public Component getTreeCellRendererComponent(JTree tree, Object value,
                        boolean selected,
                        boolean expanded,
                        boolean leaf, int row,
                        boolean hasFocus) {
    String newvalue = value.getClass().getName();    // Тип компонента
    String name = ((Component)value).getName();      // Имя компонента
    if (name != null) newvalue += " (" + name + ")"; // пока не null
    // Применяем "отображатель" для выполнения действительной работы
    return renderer.getTreeCellRendererComponent(tree, newvalue,
                          selected, expanded,
                          leaf, row, hasFocus);
  }
}

/**
 * Этот метод main() демонстрирует применение класса ComponentTree: он
 * помещает компонент ComponentTree в окно Frame и применяет ComponentTree
 * для отображения иерархии собственного GUI. Он добавляет также
 * TreeSelectionListener для отображения дополнительной информации
 * о выбранном компоненте
 **/
public static void main(String[] args) {
```

```
        // Создаем демонстрационное окно и обрабатываем запросы на закрытие окна
        JFrame frame = new JFrame("ComponentTree Demo");
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) { System.exit(0); }
          });

        // Создаем прокручиваемую панель и "строку сообщений" и размещаем их
        // по центру и внизу окна, соответственно.
        JScrollPane scrollpane = new JScrollPane();
        final JLabel msgline = new JLabel(" ");
        frame.getContentPane().add(scrollpane, BorderLayout.CENTER);
        frame.getContentPane().add(msgline, BorderLayout.SOUTH);

        // Теперь создаем объект ComponentTree, задавая окно верхнего уровня
        // в качестве компонента, дерево которого должно быть отображено.
        // Устанавливаем также шрифт для дерева.
        JTree tree = new ComponentTree(frame);
        tree.setFont(new Font("SansSerif", Font.BOLD, 12));

        // Допускается одновременный выбор только одной вершины
        tree.getSelectionModel().setSelectionMode(
                  TreeSelectionModel.SINGLE_TREE_SELECTION);

        // Добавляем слушатель, уведомляемый
        // об изменениях состояния выбора в дереве.
        tree.addTreeSelectionListener(new TreeSelectionListener() {
            public void valueChanged(TreeSelectionEvent e) {
              // Выборы в дереве описываются "путем" (path)
              // Нас интересует только последняя вершина пути
              TreePath path = e.getPath();
              Component c = (Component) path.getLastPathComponent();
              // Теперь мы знаем, какой компонент выбран, так что
              // отобразим кое-какую информацию о нем в строке сообщений
              if (c.isShowing()) {
                Point p = c.getLocationOnScreen();
                msgline.setText("x: " + p.x + " y: " + p.y +
                        " ширина: " + c.getWidth() +
                        " высота: " + c.getHeight());
              }
              else {
                msgline.setText("component is not showing");
              }
            }
          });

        // Теперь, установив дерево, поместим его в прокручиваемую панель
        scrollpane.setViewportView(tree);

        // Наконец, устанавливаем размеры главного окна и показываем его.
        frame.setSize(600, 400);
        frame.setVisible(true);
      }
    }
```

Простой веб-броузер

В двух предыдущих примерах показаны мощные компоненты JTable и JTree. Третий мощный компонент Swing – это javax.swing.text.JTextComponent и его различные подклассы, среди которых JTextField, JtextArea и JEditorPane. Компонент JEditorPane особенно интересен тем, что позволяет удобно отображать (или редактировать) HTML-текст.

Здесь стоит заметить, что для отображения статического HTML-текста нет необходимости создавать JEditorPane. Начиная с Java 1.2.2 JLabel, Jbutton и другие подобные компоненты могут отображать многострочные, многошрифтовые форматированные надписи на HTML. Весь фокус тут в том, чтобы начать надпись со строки «<html>». Это сразу скажет компоненту, чтобы он воспринимал всю надпись как форматированный HTML-текст и соответственно его отображал (используя внутренний компонент JTextComponent). С этой возможностью можно поэкспериментировать, применив программу ShowComponent; используйте ее для создания компонента JButton и установите для его свойства text значение, начинающееся с «<html>».

Пример 10.21 – это листинг *WebBrowser.java* подкласса Jframe, реализующего простой веб-броузер, показанный на рис. 10.17. Класс WebBrowser использует возможности класса java.net.URL для получения HTML-документов из Сети и компонент JEditorPane для отображения этих документов. Сконструированный как компонент, доступный для других приложений, класс WebBrowser все же содержит метод main() и, значит, может быть запущен как самостоятельное приложение.

Цель примера 10.21 – продемонстрировать богатые возможности компонента JEditorPane. Истина, однако, в том, что применение JEditorPane весьма тривиально: следует просто передать URL методу setPage() или строку HTML-текста методу setText(). Поэтому, изучая коды этого примера, не сосредоточивайтесь исключительно на JEditorPane. Правильнее будет рассматривать WebBrowser как пример соединения множества компонентов Swing и приемов программирования для создания весьма содержательного GUI. Здесь достойны внимания разрешение и запрещение объектов Action и применение компонента JFileChooser. Пример содержит также JLabel в качестве строки сообщений приложения и использует javax.swing.Timer для создания простой текстовой анимации в этой строке сообщений.

Об этом примере следует еще сказать, что он демонстрирует несколько классов из примеров, разработанных ниже в этой главе. Первый из них – это GUIResourceBundle, разработанный в примере 10.22. Этот класс позволяет считывать стандартные ресурсы GUI (такие как цвета и шрифты) из файла свойств, где они представлены в текстовом виде, что обеспечивает возможность настройки и локализации ресурсов. Дополненный реализациями ResourceParser, класс GUIResourceBundle способен потреблять более сложные «ресурсы», такие как целые компо-

Рис. 10.17. *Компонент WebBrowser*

ненты `JMenuBar` и `JToolBar`. Класс `WebBrowser` уступает право создания своих меню и панелей инструментов классу `GUIResourceBundle`.

Класс `WebBrowser` использует принимаемый по умолчанию стиль Metal, но позволяет пользователю выбрать «тему» («theme», комбинацию цвета и шрифта) в пределах этого стиля. Эта возможность обеспечивается классом `ThemeManager`, разработанным в примере 10.28. Возможности печати, имеющиеся у броузера, обеспечиваются классом `PrintableDocument`, разработанным в главе 12.

Пример 10.21. WebBrowser.java

```
package com.davidflanagan.examples.gui;
import java.awt.*;           // LayoutManager и проч.
import javax.swing.*;        // Компоненты Swing
import java.awt.event.*;     // обработчики событий AWT
import javax.swing.event.*;  // обработчики событий Swing
import java.beans.*;         // обработчики событий JavaBeans
import java.awt.print.*;     // Функции печати
import java.io.*;            // Ввод/вывод
import java.net.*;           // Сетевые операции с URL
```

```
import java.util.*;          // Хеш-таблицы и прочие служебные программы
// Импортируем этот класс по имени. Его использует JFileChooser, и его имя
// конфликтует с java.io.FileFilter
import javax.swing.filechooser.FileFilter;
// Импортируем класс для печати Swing-документов. См. главу о печати.
import com.davidflanagan.examples.print.PrintableDocument;

/**
 * Этот класс реализует простой броузер, использующий способность
 * компонента JEditorPane отображать HTML.
 **/
public class WebBrowser extends JFrame
  implements HyperlinkListener, PropertyChangeListener
{
  /**
   * Простой метод main(), позволяющий использовать класс WebBrowser
   * как самостоятельное приложение.
   **/
  public static void main(String[] args) throws IOException {
    // Прекращаем выполнение программы, когда не остается открытых
    // окон броузера
    WebBrowser.setExitWhenLastWindowClosed(true);
    WebBrowser browser = new WebBrowser(); // Создаем окно броузера
    browser.setSize(800, 600);             // Устанавливаем его размер
    browser.setVisible(true);              // Делаем его видимым.

    // Сообщаем броузеру, что ему отображать. Этот метод определен ниже.
    browser.displayPage((args.length > 0) ? args[0] : browser.getHome());
  }

  // Этот класс использует GUIResourceBundle для создания полосы меню
  // и панели инструментов. Этот статический инициализатор выполняет
  // регистрацию сразу всех необходимых классов ResourceParser.
  static {
    GUIResourceBundle.registerResourceParser(new MenuBarParser());
    GUIResourceBundle.registerResourceParser(new MenuParser());
    GUIResourceBundle.registerResourceParser(new ActionParser());
    GUIResourceBundle.registerResourceParser(new CommandParser());
    GUIResourceBundle.registerResourceParser(new ToolBarParser());
  }

  // Это компоненты Swing, используемые броузером
  JEditorPane textPane;     // Здесь отображается HTML
  JLabel messageLine;       // Здесь отображаются однострочные сообщения
  JTextField urlField;      // Здесь отображается и редактируется текущий URL
  JFileChooser fileChooser; // Позволяет пользователю выбрать локальный файл

  // Это действия Actions, используемые в полосе меню и панели инструментов.
  // Явные ссылки на них получаются из GUIResourceBundle,
  // поэтому их можно разрешать и запрещать.
  Action backAction, forwardAction;

  // В этих полях хранится история переходов для данного окна
  java.util.List history = new ArrayList(); // Список переходов
```

```
int currentHistoryPage = -1;              // Текущее положение здесь
public static final int MAX_HISTORY = 50; // За этими пределами список
                                          // обрезается

// Эти статические поля управляют поведением действия close()
static int numBrowserWindows = 0;
static boolean exitWhenLastWindowClosed = false;

// Это то место, куда нас переносит метод "home()". См. также setHome()
String home = "http://www.davidflanagan.com"; // Принимаемое по умолчанию
                                              // значение

/** Создаем и инициализируем окно WebBrowser */
public WebBrowser() {
  super();                          // Ссылка на конструктор JFrame

  textPane = new JEditorPane();  // Создаем окно для отображения HTML
  textPane.setEditable(false);   // Не позволяем пользователю
                                 // редактировать в нем

  // Регистрируем слушатели действий. Первый - для обработки гиперссылок.
  // Второй - для получения уведомлений об изменении свойств. Он сообщает
  // о том, что документ загружен. Этот класс реализует интерфейсы
  // EventListener и методы, определенные ниже
  textPane.addHyperlinkListener(this);
  textPane.addPropertyChangeListener(this);

  // Размещаем текстовую панель в JScrollPane в центре окна
  this.getContentPane().add(new JScrollPane(textPane),
              BorderLayout.CENTER);

  // Теперь создаем строку сообщений и размещаем ее внизу окна
  messageLine = new JLabel(" ");
  this.getContentPane().add(messageLine, BorderLayout.SOUTH);

  // Читаем файл WebBrowserResources.properties (и его локализованные
  // варианты, соответствующие текущему региону (locale)) для создания
  // GUIResourceBundl, из которого мы получим полосу меню
  // и панель инструментов.
  GUIResourceBundle resources =
    new GUIResourceBundle(this,"com.davidflanagan.examples.gui." +
              "WebBrowserResources");

  // Считываем полосу меню из набора ресурсов и отображаем ее
  JMenuBar menubar = (JMenuBar) resources.getResource("menubar",
                           JMenuBar.class);
  this.setJMenuBar(menubar);

 // Считываем панель инструментов из набора ресурсов. Пока не отображаем ее.
  JToolBar toolbar =
    (JToolBar) resources.getResource("toolbar", JToolBar.class);

  // Создаем текстовое поле, в которое пользователь будет вводить URL.
  // Устанавливаем слушатель действий для ответа на нажатие
  // клавиши 'ENTER' в этом поле
  urlField = new JTextField();
```

```
    urlField.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          displayPage(urlField.getText());
        }
      });

    // Помещаем поле URL и надпись к нему в конец панели инструментов
    toolbar.add(new JLabel("          URL:"));
    toolbar.add(urlField);

    // И размещаем панель инструментов вверху окна
    this.getContentPane().add(toolbar, BorderLayout.NORTH);

    // Считываем из набора ресурсов кэшированные копии двух объектов Action
    // Эти действия используются полосой меню и панелью инструментов.
    // Их разрешение и запрещение разрешает и запрещает полосу меню
    // и панель инструментов.
    backAction = (Action)resources.getResource("action.back",Action.class);
    forwardAction =
      (Action)resources.getResource("action.forward", Action.class);

    // Стартуем с запрещенными действиями
    backAction.setEnabled(false);
    forwardAction.setEnabled(false);

    // Создаем ThemeManager для этого окна,
    // и помещаем меню Theme в полосу меню
    ThemeManager themes = new ThemeManager(this, resources);
    menubar.add(themes.getThemeMenu());

    // Отслеживаем число открытых окон броузера
    WebBrowser.numBrowserWindows++;
  }

  /** Устанавливаем статическое свойство, управляющее поведением close() */
  public static void setExitWhenLastWindowClosed(boolean b) {
    exitWhenLastWindowClosed = b;
  }

  /** Это методы доступа к свойству home. */
  public void setHome(String home) { this.home = home; }
  public String getHome() { return home; }

  /**
   * Этот внутренний метод пытается загрузить и отобразить заданный URL.
   * Он вызывается в разных местах класса.
   **/
  boolean visit(URL url) {
    try {
      String href = url.toString();
      // Начало анимации. Анимация останавливается в propertyChanged()
      startAnimation("Загрузка " + href + "...");
      textPane.setPage(url);  // Загружаем и отображаем URL
      this.setTitle(href);    // Отображаем URL в заголовке окна
      urlField.setText(href); // Отображаем URL в поле ввода текста
```

```
    return true;              // Докладываем об успехе
  }
  catch (IOException ex) {   // Если загрузка страницы не удалась
    stopAnimation();
   messageLine.setText("Невозможно загрузить страницу: " + ex.getMessage());
    return false;       // Сигнализируем об ошибке
  }
}

/**
 * Просим броузер отобразить заданный URL и помещаем его в историю.
 **/
public void displayPage(URL url) {
  if (visit(url)) {      // Переходим на заданный url и в случае успеха
    history.add(url);    // помещаем url в историю.
    int numentries = history.size();
    if (numentries > MAX_HISTORY+10) { // Обрезаем слишком длинный список
      history = history.subList(numentries-MAX_HISTORY, numentries);
      numentries = MAX_HISTORY;
    }
    currentHistoryPage = numentries-1; // Устанавливаем положение текущей
                                       // страницы в истории
    // Если есть куда вернуться, разрешаем действие Back
    if (currentHistoryPage > 0) backAction.setEnabled(true);
  }
}

/** Этот код делает то же, что и displayPage(URL), но принимает строку */
public void displayPage(String href) {
  try {
    displayPage(new URL(href));
  }
  catch (MalformedURLException ex) {
    messageLine.setText("Bad URL: " + href);
  }
}

/** Позволяем пользователю выбрать локальный файл и отобразить его */
public void openPage() {
  // Ленивое создание: не создаем JfileChooser до тех пор,
  // пока это не понадобится
  if (fileChooser == null) {
    fileChooser = new JFileChooser();
    // Этот фильтр javax.swing.filechooser.FileFilter отображает
    // только файлы HTML
    FileFilter filter = new FileFilter() {
        public boolean accept(File f) {
          String fn = f.getName();
          if (fn.endsWith(".html") || fn.endsWith(".htm"))
            return true;
          else return false;
        }
        public String getDescription() { return "HTML Files"; }
```

```
      };
    fileChooser.setFileFilter(filter);
    fileChooser.addChoosableFileFilter(filter);
  }

  // Просим пользователя выбрать файл.
  int result = fileChooser.showOpenDialog(this);
  if (result == JFileChooser.APPROVE_OPTION) {
    // Если он не нажал 'Cancel', пытаемся отобразить файл.
    File selectedFile = fileChooser.getSelectedFile();
    String url = "file://" + selectedFile.getAbsolutePath();
    displayPage(url);
  }
}

/** Возвращаемся к предыдущей странице. */
public void back() {
  if (currentHistoryPage > 0) // Идем назад, если можем
    visit((URL)history.get(--currentHistoryPage));
  // Соответственно разрешаем и запрещаем действия
  backAction.setEnabled((currentHistoryPage > 0));
  forwardAction.setEnabled((currentHistoryPage < history.size()-1));
}

/** Переходим (вперед) на следующую страницу истории */
public void forward() {
  if (currentHistoryPage < history.size()-1) // Идем вперед, если можем
    visit((URL)history.get(++currentHistoryPage));
  // Соответственно разрешаем и запрещаем действия
  backAction.setEnabled((currentHistoryPage > 0));
  forwardAction.setEnabled((currentHistoryPage < history.size()-1));
}

/** Перезагружаем текущую страницу */
public void reload() {
  if (currentHistoryPage != -1)
    visit((URL)history.get(currentHistoryPage));
}

/** Отображаем страницу, заданную свойством "home" */
public void home() { displayPage(getHome()); }

/** Открываем новое окно броузера */
public void newBrowser() {
  WebBrowser b = new WebBrowser();
  b.setSize(this.getWidth(), this.getHeight());
  b.setVisible(true);
}

/**
 * Закрываем это окно броузера. Если оно было единственным, и значение
 * exitWhenLastBrowserClosed равно true, тогда завершаем работу VM
 **/
public void close() {
  this.setVisible(false);           // Скрываем окно
```

```
  this.dispose();                    // Уничтожаем окно
  synchronized(WebBrowser.class) {  // Синхронизируем для обеспечения
                                     // безопасности потоков исполнения
    WebBrowser.numBrowserWindows--; // Теперь на одно окно стало меньше
    if ((numBrowserWindows==0) && exitWhenLastWindowClosed)
      System.exit(0);       // Выходим, если это было последнее окно
  }
}

/**
 * Выход из VM. Если confirm имеет значение true, запрашиваем
 * у пользователя подтверждение. Обратите внимание на то, что
 * showConfirmDialog() отображает диалог, ждет пользователя
 * и возвращает его ответ (т. е. кнопку, нажатую пользователем).
 **/
public void exit(boolean confirm) {
  if (!confirm ||
    (JOptionPane.showConfirmDialog(this, // родительский диалог
       /*отображаемое сообщение */ "Are you sure you want to quit?",
       /* название диалога */    "Really Quit?",
       /* кнопки диалога */   JOptionPane.YES_NO_OPTION) ==
     JOptionPane.YES_OPTION)) // Если нажата кнопка Yes
    System.exit(0);
}

/**
 * Печатаем содержимое текстовой панели с применением java.awt.print API
 * Обратите внимание, что этот API не эффективен в Java 1.2
 * Всю трудную работу выполняет класс PrintableDocument.
 **/
public void print() {
  // Получаем от системы объект PrinterJob
  PrinterJob job = PrinterJob.getPrinterJob();
  // Это объект, который мы будем печатать
  PrintableDocument pd = new PrintableDocument(textPane);
  // Сообщаем PrinterJob, что именно мы хотим напечатать
  job.setPageable(pd);
  // Отображаем диалог печати, спрашиваем у пользователя, какие страницы
  // печатать, на какой принтер направить печать,
  // и даем пользователю шанс отменить задание.
  if (job.printDialog()) { // Если пользователь не отменил задание
    try { job.print(); }   // Начинаем печатать!
    catch(PrinterException ex) { // красиво отображаем сообщение об ошибках
      messageLine.setText("Невозможно напечатать: " + ex.getMessage());
    }    }
}

/**
 * Этот метод реализует HyperlinkListener. Он вызывается, когда
 * пользователь щелкает на гиперссылке или помещает на ссылку указатель мыши
 **/
public void hyperlinkUpdate(HyperlinkEvent e) {
  HyperlinkEvent.EventType type = e.getEventType(); // что случилось?
```

```
  if (type == HyperlinkEvent.EventType.ACTIVATED) {   // Щелчок!
    displayPage(e.getURL());  // Идем по ссылке; отображаем новую страницу
  }
  else if (type == HyperlinkEvent.EventType.ENTERED) { // Указатель мыши
                                                       // над ссылкой!
    // Когда указатель мыши перемещается над ссылкой, отображаем
    // ее в строке сообщений
    messageLine.setText(e.getURL().toString());
  }
  else if (type == HyperlinkEvent.EventType.EXITED) { // Указатель мыши
                                                     // ушел за пределы ссылки!
    messageLine.setText(" "); // Очищаем строку сообщений
  }
}

/**
 * Этот метод реализует java.beans.PropertyChangeListener. Он вызывается,
 * когда у объекта JEditorPane изменяется связанное свойство.
 * Нас интересует свойство "page", поскольку оно сообщает
 * об окончании загрузки страницы.
 **/
public void propertyChange(PropertyChangeEvent e) {
  if (e.getPropertyName().equals("page")) // Если свойство изменилось
    stopAnimation();       // Останавливаем анимацию loading...
}

/**
 * Приведенные ниже поля и методы реализуют простую анимацию
 * в строке сообщений броузера; она используется, чтобы обеспечить
 * пользователю "обратную связь" на время загрузки страницы.
 **/
String animationMessage; // Отображаемое сообщение "loading..."
int animationFrame = 0;  // В каком "кадре" фильма мы находимся
String[] animationFrames = new String[] { // Содержимое каждого "кадра"
  "-", "\\", "|", "/", "-", "\\", "|", "/",
  ",", ".", "o", "O", "O", "#", "*", "+"
};

/** Этот объект вызывает метод animate() 8 раз в секунду */
javax.swing.Timer animator =
  new javax.swing.Timer(125, new ActionListener() {
      public void actionPerformed(ActionEvent e) { animate(); }
    });

/** Отображаем следующий кадр. Вызов из таймера анимации */
void animate() {
  String frame = animationFrames[animationFrame++];     // Получаем
                                                        // следующий кадр
  messageLine.setText(animationMessage + " " + frame); // Обновляем строку
                                                        // сообщений
  animationFrame = animationFrame % animationFrames.length;
}

/** Запускаем анимацию. Вызов из метода visit(). */
```

```
    void startAnimation(String msg) {
      animationMessage = msg;    // Сохраняем отображаемое сообщение
      animationFrame = 0;        // Начинаем с кадра 0 анимации
      animator.start();          // Запускаем таймер.
    }

    /** Останавливаем анимацию. Вызов из метода propertyChanged(). */
    void stopAnimation() {
      animator.stop();   // Приказываем таймеру прекратить выстреливать события
      messageLine.setText(" ");  // Очищаем строку сообщений
    }
  }
```

Описание GUI при помощи свойств

По существу, задача определения пользовательского интерфейса является описательной. Эта описательная задача не слишком хорошо увязывается с таким основанным на процедурах и алгоритмах языком, как Java. Приходится писать массу кода, создающего компоненты, устанавливающего свойства и помещающего компоненты в контейнеры. Вместо простого описания структуры желаемого GUI приходится писать код, чтобы шаг за шагом построить GUI.

Один из способов избежать написания этих нудных кодов построения GUI – создать некий язык описаний GUI, а затем написать программу, которая сможет прочитать этот язык и автоматически создать описанный GUI. Один из распространенных подходов – описывать GUI с применением грамматики XML. В этой главе мы будем опираться на более простой синтаксис файлов свойств Java, используемых класом `ResourceBundle`. (См. главу 7 «Интернационализация», где имеются примеры применения `java.util.ResourceBundle`.)

Объект `java.util.Properties` представляет собой хеш-таблицу, которая отображает строковые ключи на строковые значения. Класс `Properties` может читать и записывать файлы в формате, в котором каждая строка вида *`name`*`:`*`value`* определяет одно свойство. Более того, объект `Properties` может иметь родительский объект `Properties`. При поиске значения свойства, отсутствующего в данном объекте `Properties`, поиск будет продолжен в родительском объекте (и так далее – по рекурсии). Класс `ResourceBundle` образует вокруг свойств интернационализационный слой, позволяющий настраивать свойства в зависимости от местных установок. Интернационализация является важным аспектом приложений с GUI, поэтому класс `ResourceBundle` представляется подходящим для описания ресурсов GUI.

Обработка основных ресурсов GUI

Поскольку файлы свойств являются текстовыми, при работе с объектами `ResourceBundle`, основанными на файлах свойств, имеется ограни-

чение, состоящее в том, что они поддерживают только ресурсы типа `String`. Класс `GUIResourceBundle`, представленный в примере 10.22, является подклассом `ResourceBundle`, который добавляет методы для чтения строковых ресурсов и преобразования их в объекты типов, широко представленных в программировании GUI, таких как `Color` и `Font`.

Код `GUIResourceBundle` совершенно прозрачен. Интерфейс `ResourceParser` обеспечивает механизм расширения; он будет рассмотрен в следующем примере. Обратите внимание на то, что класс `MalformedResourceException`, используемый в этом примере, не является стандартным классом Java; это самостоятельно определяемый подкласс `MissingResourceException`, разработанный специально для этого примера. Поскольку это тривиальный подкласс, его код здесь не приводится, но его можно найти в онлайновом архиве примеров.

Пример 10.22. GUIResourceBundle.java

```
package com.davidflanagan.examples.gui;
import java.io.*;
import java.util.*;
import java.awt.*;

/**
 * Этот класс конкретизирует ResourceBundle и добавляет методы для получения
 * ресурсов типов, часто используемых в GUI. К тому же он вносит
 * расширяемость, позволяя регистрироваться объектам ResourceParser,
 * способным разбирать ресурсы других типов.
 **/
public class GUIResourceBundle extends ResourceBundle {
  // Корневой объект. Требуется для анализа ресурсов некоторых
  // типов, например Command
  Object root;

  // Набор ресурсов, который фактически содержит текстовые ресурсы
  // Этот класс является оболочкой (wrapper) для этого набора
  ResourceBundle bundle;

  /** Создаем оболочку GUIResourceBundle вокруг заданного набора */
  public GUIResourceBundle(Object root, ResourceBundle bundle) {
    this.root = root;
    this.bundle = bundle;
  }

  /**
   * Загружаем набор, заданный по имени, и создаем GUIResourceBundle вокруг
   * него. Этот конструктор использует возможности интернационализации,
   * имеющиеся у метода ResourceBundle.getBundle().
   **/
  public GUIResourceBundle(Object root, String bundleName)
    throws MissingResourceException
  {
    this.root = root;
    this.bundle = ResourceBundle.getBundle(bundleName);
  }
```

```
/**
 * Создаем PropertyResourceBundle из заданного потока (stream), а затем
 * создаем для него оболочку GUIResourceBundle
 **/
public GUIResourceBundle(Object root, InputStream propertiesStream)
  throws IOException
{
  this.root = root;
  this.bundle = new PropertyResourceBundle(propertiesStream);
}

/**
 * Создаем PropertyResourceBundle из заданного файла свойств, и затем
 * создаем для него оболочку GUIResourceBundle.
 **/
public GUIResourceBundle(Object root, File propertiesFile)
  throws IOException
{
  this(root, new FileInputStream(propertiesFile));
}

/** Это один из абстрактных методов ResourceBundle */
public Enumeration getKeys() { return bundle.getKeys(); }

/** Это еще один абстрактный метод ResourceBundle */
protected Object handleGetObject(String key)
  throws MissingResourceException
{
  return bundle.getObject(key); // Просто вызываем метод оболочки
}

/** Это метод доступа к свойствам нашего корневого объекта */
public Object getRoot() { return root; }

/**
 * Этот метод похож на унаследованный метод getString() за исключением
 * того, что когда названный ресурс не найден, метод возвращает принимаемый
 * по умолчанию ресурс, а не выдает исключение
 **/
public String getString(String key, String defaultValue) {
  try { return bundle.getString(key); }
  catch(MissingResourceException e) { return defaultValue; }
}

/**
 * Ищем названный ресурс и преобразуем его в список строк, разделенных
 * пробелами, символами табуляции или запятыми.
 **/
public java.util.List getStringList(String key)
  throws MissingResourceException
{
  String s = getString(key);
  StringTokenizer t = new StringTokenizer(s, ", \t", false);
```

```
  ArrayList list = new ArrayList();
  while(t.hasMoreTokens()) list.add(t.nextToken());
  return list;
}

/** Как выше, но вместо выдачи исключения возвращаем принимаемый
    по умолчанию ресурс */
public java.util.List getStringList(String key,
                  java.util.List defaultValue) {
  try { return getStringList(key); }
  catch(MissingResourceException e) { return defaultValue; }
}

/** Ищем названный ресурс и пытаемся проинтерпретировать его
    как логическое значение. */
public boolean getBoolean(String key) throws MissingResourceException {
  String s = bundle.getString(key);
  s = s.toLowerCase();
  if (s.equals("true")) return true;
  else if (s.equals("false")) return false;
  else if (s.equals("yes")) return true;
  else if (s.equals("no")) return false;
  else if (s.equals("on")) return true;
  else if (s.equals("off")) return false;
  else {
    throw new MalformedResourceException("boolean", key);
  }
}

/** Как выше, но вместо выдачи исключения возвращаем принимаемое
    по умолчанию значение */
public boolean getBoolean(String key, boolean defaultValue) {
  try { return getBoolean(key); }
  catch(MissingResourceException e) {
    if (e instanceof MalformedResourceException)
      System.err.println("WARNING: " + e.getMessage());
    return defaultValue;
  }
}

/** То же, что getBoolean(), но для целых */
public int getInt(String key) throws MissingResourceException {
  String s = bundle.getString(key);

  try {
    // Применяя decode() вместо parseInt(), мы поддерживаем восьмеричные
    // и шестнадцатеричные представления
    return Integer.decode(s).intValue();
  } catch (NumberFormatException e) {
    throw new MalformedResourceException("int", key);
  }
}

/** Как выше, но с принимаемым по умолчанию значением */
```

```
public int getInt(String key, int defaultValue) {
  try { return getInt(key); }
  catch(MissingResourceException e) {
    if (e instanceof MalformedResourceException)
      System.err.println("WARNING: " + e.getMessage());
    return defaultValue;
  }
}

/** Возвращаем ресурс типа double */
public double getDouble(String key) throws MissingResourceException {
  String s = bundle.getString(key);

  try {
    return Double.parseDouble(s);
  } catch (NumberFormatException e) {
    throw new MalformedResourceException("double", key);
  }
}

/** Как выше, но с принимаемым по умолчанию значением */
public double getDouble(String key, double defaultValue) {
  try { return getDouble(key); }
  catch(MissingResourceException e) {
    if (e instanceof MalformedResourceException)
      System.err.println("WARNING: " + e.getMessage());
    return defaultValue;
  }
}

/** Ищем ресурс с заданным именем и преобразуем его в Font */
public Font getFont(String key) throws MissingResourceException {
  // Метод Font.decode() всегда возвращает объект Font,
  // поэтому мы не можем проверить, правильно ли записан ресурс.
  return Font.decode(bundle.getString(key));
}

/** Как выше, но с принимаемым по умолчанию значением */
public Font getFont(String key, Font defaultValue) {
  try { return getFont(key); }
  catch (MissingResourceException e) { return defaultValue; }
}

/** Ищем ресурс с заданным именем и преобразуем его в Color */
public Color getColor(String key) throws MissingResourceException {
  try {
    return Color.decode(bundle.getString(key));
  }
  catch (NumberFormatException e) {
    // Было бы полезно попробовать здесь разбирать названия цветов
    // наряду с числовыми спецификациями цветов
    throw new MalformedResourceException("Color", key);
  }
}
```

```
/** Как выше, но с принимаемым по умолчанию значением */
public Color getColor(String key, Color defaultValue) {
  try { return getColor(key); }
  catch(MissingResourceException e) {
    if (e instanceof MalformedResourceException)
      System.err.println("WARNING: " + e.getMessage());
    return defaultValue;
  }
}

/** Хеш-таблица, связывающая типы ресурсов и их анализаторы (parser) */
static HashMap parsers = new HashMap();

/** Механизм расширения: регистрирует анализатор для нового типа ресурса */
public static void registerResourceParser(ResourceParser parser) {
  // Спрашиваем у ResourceParser, какие типы он понимает
  Class[] supportedTypes = parser.getResourceTypes();
  // Регистрируем его в хеш-таблице для каждого типа
  for(int i = 0; i < supportedTypes.length; i++)
    parsers.put(supportedTypes[i], parser);
}

/** Ищем ResourceParser для заданного типа ресурса */
public static ResourceParser getResourceParser(Class type) {
  return (ResourceParser) parsers.get(type);
}

/**
 * Ищем ResourceParser заданного типа и, если находим,
 * просим его разобрать и вернуть заданный по имени ресурс
 **/
public Object getResource(String key, Class type)
  throws MissingResourceException
{
  // Получаем анализатор для заданного типа
  ResourceParser parser = (ResourceParser)parsers.get(type);
  if (parser == null)
    throw new MissingResourceException(
       " Объект ResourceParser не зарегистрирован для " +
       type.getName() + " resources",
       type.getName(), key);

  try { // Просим анализатор разобрать ресурс
    return parser.parse(this, key, type);
  }
  catch(MissingResourceException e) {
    throw e; // Передаем выше исключение MissingResourceException
  }
  catch(Exception e) {
    // Если выдается любое другое исключение, оно
    // преобразуется в MalformedResourceException
    String msg = "Malformed " + type.getName() + " resource: " +
      key + ": " + e.getMessage();
```

```
            throw new MalformedResourceException(msg, type.getName(), key);
        }
    }

    /**
     * То же, что двухаргументный getResource, но возвращает принимаемое
     * по умолчанию значение вместо исключения MissingResourceException
     **/
    public Object getResource(String key, Class type, Object defaultValue) {
        try { return getResource(key, type); }
        catch (MissingResourceException e) {
            if (e instanceof MalformedResourceException)
                System.err.println("WARNING: " + e.getMessage());
            return defaultValue;
        }
    }
}
```

Механизм расширения для сложных ресурсов

Как мы только что видели, пример 10.22 использует интерфейс `ResourceParser` **для создания механизма расширения, позволяющего ему обрабатывать большее число типов ресурсов. Пример 10.23 – листинг этого простого интерфейса. Мы увидим некоторые интересные реализации этого интерфейса в следующих разделах.**

Пример 10.23. ResourceParser.java

```
package com.davidflanagan.examples.gui;

/**
 * Этот интерфейс определяет механизм расширения,
 * позволяющий GUIResourceBundle разбирать ресурсы произвольных типов
 **/
public interface ResourceParser {
    /**
     * Возвращает массив классов, указывающий, какие ресурсы
     * этот анализатор умеет обрабатывать
     **/
    public Class[] getResourceTypes();

    /**
     * Считывет свойство, заданное ключом key, из заданного набора bundle,
     * преобразует его к заданному типу type и возвращает свойство. Для сложных
     * ресурсов анализатору может понадобиться считать из набора больше одного
     * свойства; обычно это множество свойств, названия которых начинаются
     * с заданного ключа.
     **/
    public Object parse(GUIResourceBundle bundle, String key, Class type)
        throws Exception;
}
```

Анализ команд и действий

В качестве первой реализации интерфейса ResourceParser мы добавим возможность разбора объектов Action. Как мы уже видели, объекты Action часто применяются в GUI; объекты Action содержат множество атрибутов, таких как описание, графический символ и всплывающая подсказка, которые могут нуждаться в локализации. Наша реализация ActionParser основывается на показанном в примере 10.17 классе CommandAction, который, в свою очередь, опирается на связанные с отражением возможности показанного в примере 8.2 класса Command.

Чтобы реализовать класс ActionParser, необходимо уметь получать объекты Command из файла свойств. Начнем поэтому с класса CommandParser, показанного в примере 10.24. Этот класс весьма прост, так как он опирается на аналитические способности класса Command. Листинг класса ActionParser следует ниже, в примере 10.25.

Чтобы было легче понять, как работает этот класс, рассмотрим следующие свойства, извлеченные из файла *WebBrowserResources.properties*, используемого классом WebBrowser из примера 10.21:

```
action.home: home();
action.home.label: Home
action.home.description: Go to home page
action.oreilly: displayPage("http://www.oreilly.com");
action.oreilly.label: O'Reilly
action.oreilly.description: O'Reilly & Associates home page
```

Эти свойства описывают два действия, задаваемые ключами «action.home» и «action.oreilly».

Пример 10.24. CommandParser.java

```
package com.davidflanagan.examples.gui;
import com.davidflanagan.examples.reflect.Command;

/**
 * Этот класс получает объект Command из GUIResourceBundle. Он использует
 * метод Command.parse() для выполнения действительной работы.
 **/
public class CommandParser implements ResourceParser {
  static final Class[] supportedTypes = new Class[] { Command.class };
  public Class[] getResourceTypes() { return supportedTypes;}

  public Object parse(GUIResourceBundle bundle, String key, Class type)
    throws java.util.MissingResourceException, java.io.IOException
  {
    String value = bundle.getString(key); // ищем текст команды
    return Command.parse(bundle.getRoot(), value); // разбираем и получаем ее!
  }
}
```

Пример 10.25. ActionParser.java

```
package com.davidflanagan.examples.gui;
import com.davidflanagan.examples.reflect.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

/**
 * Этот класс получает объект Action из GUIResourceBundle.
 * Заданный ключ используется для поиска строки Command для этого действия.
 * Этот ключ используется также в качестве приставки к именам ресурсов,
 * задающих другие атрибуты этого действия (такие как надпись и значок).
 * Действие «zoomOut» может задаваться, например, так:
 *
 *   zoomOut: zoom(0.5);
 *   zoomOut.label: Zoom Out
 *   zoomOut.description: Zoom out by a factor of 2
 *
 * Поскольку объекты Action часто повторно используются в приложении (например,
 * в панели инструментов и в системе меню), этот ResourceParser кэширует
 * возвращаемые им объекты Action. За счет совместного использования
 * объектов Action разрешение и запрещение действия может оказывать
 * влияние на GUI в целом.
 **/
public class ActionParser implements ResourceParser {
  static final Class[] supportedTypes = new Class[] { Action.class };
  public Class[] getResourceTypes() { return supportedTypes; }

  HashMap bundleToCacheMap = new HashMap();

  public Object parse(GUIResourceBundle bundle, String key, Class type)
    throws java.util.MissingResourceException
  {
    // Ищем кэш с объектами Action, ассоциированными с этим набором
    HashMap cache = (HashMap) bundleToCacheMap.get(bundle);
    if (cache == null) { // Если кэш отсутствует, создаем и сохраняем его
      cache = new HashMap();
      bundleToCacheMap.put(bundle, cache);
    }
    // Теперь ищем в кэше объект Action, ассоциированный с данным ключом.
    Action action = (Action) cache.get(key);
    // Если находим кэшированное действие, возвращаем его.
    if (action != null) return action;

    // Если кэшированное действие не обнаруживается, создаем его.
    // Единственным нужным ресурсом является команда.
    // При ее отсутствии или неправильном формате будет выдано исключение.
    Command command = (Command) bundle.getResource(key, Command.class);

    // Далее следуют вызовы, выдающие принимаемые по умолчанию значения,
    // поэтому они не выдают исключений, даже если объекты ResourceParser
    // не были зарегистрированы для таких типов, как Icon или KeyStroke
    String label = bundle.getString(key + ".label", null);
```

```
    Icon icon = (Icon) bundle.getResource(key + ".icon", Icon.class, null);
    String tooltip = bundle.getString(key + ".description", null);
    KeyStroke accelerator =
      (KeyStroke) bundle.getResource(key + ".accelerator",
                     KeyStroke.class, null);
    int mnemonic = bundle.getInt(key + ".mnemonic", KeyEvent.VK_UNDEFINED);
    boolean enabled = bundle.getBoolean(key + ".enabled", true);

    // Создаем объект CommandAction на основе этих значений
    action = new CommandAction(command, label, icon, tooltip,
                  accelerator, mnemonic, enabled);

    // Сохраняем его в кэше, затем возвращаем
    cache.put(key, action);
    return action;
  }
}
```

Анализ меню

Мы видели, как класс `GUIResourceBundle` облегчает считывание из файла свойств таких простых ресурсов GUI, как цвет и шрифт. Мы видели также, как можно расширить `GUIResourceBundle`, чтобы научиться анализировать более сложные ресурсы, подобные объектам `Action`. Шрифты, цвета и действия – это ресурсы, которые используют составляющие GUI компоненты. Совершив небольшой скачок в понятиях, мы можем начать думать о самих компонентах GUI как о ресурсах, которые использует некое большее приложение.

Примеры 10.26 и 10.27 показывают, как это можно сделать. В этих примерах приводятся определения классов `MenuBarParser` и `MenuParser`, считывающих из файла свойств объекты `JMenuBar` и `JMenu` соответственно. `MenuBarParser` полагается на `MenuParser` в том, что касается получения объектов `JMenu`, населяющих полосу меню, а `MenuParser` поручает приведенному выше классу `ActionParser` получить объекты `Action`, представляющие элементы меню в каждом `JMenu`.

`MenuParser` и `MenuBarParser` считывают из файла свойств описания меню, используя простую грамматику, которую илюстрируют следующие строки из файла *WebBrowserResource.properties*:

```
# Полоса меню содержит два меню, "menu.file" и "menu.go"
menubar: menu.file menu.go

# Меню "menu.file" имеет надпись "File" и содержит пять элементов,
# заданных как объекты действий. Эти элементы поделены разделителем
# на две группы
menu.file: File: action.new action.open action.print - action.close action.exit

# Меню "menu.go" имеет надпись "Go" и содержит четыре элемента
menu.go: Go: action.back action.forward action.reload action.home
```

В этих строках описываются полоса меню с именем свойства «menubar» и все ее подменю. Обратите внимание, что здесь опущены свойства, определяющие отдельные действия, входящие в каждое меню.

Как вы можете видеть, грамматика полосы меню весьма проста: в ней просто перечисляются названия меню, входящих в полосу меню. По этой причине и код `MenuBarParser` в примере 10.26 очень прост. Грамматика описаний меню несколько сложнее, что отражено в примере 10.27.

Можно вспомнить, что пример `WebBrowser` также использует `GUIResourceBundle` для считывания `JToolBar` из файла свойств. Это делается с применением класса `ToolBarParser`. Код этого класса очень похож на код `MenuBarParser` и не приводится здесь. Он, однако, доступен в онлайновом архиве примеров.

Пример 10.26. MenuBarParser.java

```
package com.davidflanagan.examples.gui;
import javax.swing.*;
import java.util.*;

/**
 * Получаем JMenuBar из ResourceBundle. Полоса меню представляется
 * просто как список свойств меню. Например,
 *   menubar: menu.file menu.edit menu.view menu.help
 **/
public class MenuBarParser implements ResourceParser {
  static final Class[] supportedTypes = new Class[] { JMenuBar.class };
  public Class[] getResourceTypes() { return supportedTypes; }

  public Object parse(GUIResourceBundle bundle, String key, Class type)
    throws java.util.MissingResourceException
  {
    // Получаем значение ключа в виде списка строк
    List menuList = bundle.getStringList(key);

    // Создаем MenuBar
    JMenuBar menubar = new JMenuBar();

    // Создаем JMenu для каждого меню и помещаем его в полосу
    int nummenus = menuList.size();
    for(int i = 0; i < nummenus; i++) {
      menubar.add((JMenu) bundle.getResource((String)menuList.get(i),
                          JMenu.class));
    }

    return menubar;
  }
}
```

Пример 10.27. MenuParser.java

```
package com.davidflanagan.examples.gui;
import com.davidflanagan.examples.reflect.*;
import java.awt.event.*;
```

```
import javax.swing.*;
import java.util.StringTokenizer;

/**
 * Этот класс получает JMenu или JPopupMenu по текстовому описанию,
 * найденному в GUIResourceBundle. Грамматика простая: надпись меню,
 * за которой следуют двоеточие и список элементов меню. Элементы меню,
 * начинающиеся с символа '>', являются подменю. Элементы меню,
 * начинающиеся с символа '-', являются разделителями.
 * Все другие элементы являются именами действий.
 **/
public class MenuParser implements ResourceParser {
  static final Class[] supportedTypes = new Class[] {
    JMenu.class, JPopupMenu.class // Этот класс обрабатывает ресурсы двух типов
  };

  public Class[] getResourceTypes() { return supportedTypes; }

  public Object parse(GUIResourceBundle bundle, String key, Class type)
    throws java.util.MissingResourceException
  {
    // Получаем строковое значение ключа
    String menudef = bundle.getString(key);

    // Разбиваем его на слова, игнорируя пробелы, двоеточия и запятые
    StringTokenizer st = new StringTokenizer(menudef, " \t:,");

    // Первое слово - это надпись меню
    String menuLabel = st.nextToken();

    // Создаем либо Jmenu, либо JPopupMenu
    JMenu menu = null;
    JPopupMenu popup = null;
    if (type == JMenu.class) menu = new JMenu(menuLabel);
    else popup = new JPopupMenu(menuLabel);

    // Теперь в цикле проходим все остальные слова, создавая для каждого
    // JmenuItem. Накапливаем эти элементы в списке
    while(st.hasMoreTokens()) {
      String item = st.nextToken();    // следующее слово
      char firstchar = item.charAt(0); // определяем тип элемента меню
      switch(firstchar) {
      case '-':  // слова, начинающиеся с '-', добавляют в меню разделитель
        if (menu != null) menu.addSeparator();
        else popup.addSeparator();
        break;
      case '>':  // слова, начинающиеся с '>', являются именами подменю
        // отрезаем символ '>' и рекурсивно обрабатываем подменю
        item = item.substring(1);
        // Разбираем подменю и помещаем его в список элементов
        JMenu submenu = (JMenu)parse(bundle, item, JMenu.class);
        if (menu != null) menu.add(submenu);
        else popup.add(submenu);
        break;
```

```
        case '!': // слова, начинающиеся с '!', - это названия действий
  item = item.substring(1);  // отрезаем символ '!'
          /* проваливаемся */     // проваливаемся к следующему действию
        default: // По умолчанию все другие слова являются названиями действий
          // Ищем действие с этим названием и помещаем его в меню
          Action action = (Action)bundle.getResource(item, Action.class);
          if (menu != null) menu.add(action);
          else popup.add(action);
          break;
        }
      }

      // Наконец, возвращаем меню или всплывающее меню
      if (menu != null) return menu;
      else return popup;
    }
  }
```

Темы и стиль Metal

Принимаемый по умолчанию платформенно-независимый стиль приложений Swing известен как стиль Metal. Одна из сильных, но малоизвестных черт стиля Metal состоит в том, что используемые им цвета и шрифты легко поддаются настройке. Все, что нужно сделать, – это передать объект `MetalTheme` статическому методу `setCurrentTheme()` класса `MetalLookAndFeel`. (Эти классы определены в редко используемом пакете `javax.swing.plaf.metal`.)

Класс `MetalTheme` является абстрактным, поэтому на практике приходится работать с классом `DefaultMetalTheme`. У этого класса есть шесть методов, возвращающих цвета основных тем (на самом деле по три оттенка основного и побочного цветов), и четыре метода, возвращающих шрифты основных тем. Чтобы определить новую тему, нужно только создать подкласс `DefaultMetalTheme` и заместить эти методы, чтобы они возвращали желательные цвета и шрифты. (Если нужна более глубокая переделка, придется образовать подкласс прямо от `MetalTheme`.)

Пример 10.28 – это листинг *ThemeManager.java*. Этот пример включает в себя подкласс `DefaultMetalTheme`, но определяет его как внутренний класс `ThemeManager`. Класс `ThemeManager` предоставляет возможность считывать определения тем (то есть спецификации шрифта и цвета) из набора `GUIResourceBundle`. Он также определяет методы для чтения имени темы, принимаемой по умолчанию, и списка имен всех доступных тем набора. Наконец, `ThemeManager` может возвращать компонент `Jmenu`, отображающий для пользователя список доступных тем, и переключать текущую тему на основе выбора, сделанного пользователем.

Класс `ThemeManager` и компонент `Jmenu`, который он создает, использованы в классе `WebBrowser` из примера 10.21. Перед изучением кода `ThemeManager` взгляните на извлеченные из файла *WebBrowserResources.pro-*

perties строки, которые определяют множество доступных для веб-браузера тем:

```
# Это свойство определяет имена свойств для всех доступных тем.
themelist: theme.metal theme.rose, theme.lime, theme.primary, theme.bigfont
# Это свойство определяет имя принимаемого по умолчанию свойства
defaultTheme: theme.metal

# У этой темы есть только имя. Все шрифты и значения цветов не отличаются от
# принимаемой по умолчанию темы стиля Metal
theme.metal.name: Default Metal

# Эта тема использует оттенки красного и розового
theme.rose.name: Rose
theme.rose.primary: #905050
theme.rose.secondary: #906050

# Эта тема использует желто-зеленые цвета
theme.lime.name: Lime
theme.lime.primary: #509050
theme.lime.secondary: #506060

# Эта тема использует яркие основные цвета
theme.primary.name: Primary Colors
theme.primary.primary: #202090
theme.primary.secondary: #209020

# Эта тема использует крупный шрифт и цвета, принимаемые по умолчанию
theme.bigfont.name: Big Fonts
theme.bigfont.controlFont: sansserif-bold-18
theme.bigfont.menuFont: sansserif-bold-18
theme.bigfont.smallFont: sansserif-plain-14
theme.bigfont.systemFont: sansserif-plain-14
theme.bigfont.userFont: sansserif-plain-14
theme.bigfont.titleFont: sansserif-bold-18
```

Глядя на эти определения тем, можно без труда понять код `ThemeManager`. Метод `getThemeMenu()` создает компонент `Jmenu`, населенный объектами `JRadioButtonMenuItem`, а не объектами `JMenuItem` или `Action`, как это делалось выше в этой главе. Это подчеркивает тот факт, что в каждый момент времени может быть выбрана только одна тема. Когда тема меняется, метод `setTheme()` использует метод из `SwingUtilities` для распространения этой темы на все компоненты в окне. Наконец, обратите внимание на то, что внутренний класс `Theme` вместо объектов `Font` и `Color` использует объекты `FontUIResource` и `ColorUIResource`. Эти классы входят в пакет `javax.swing.plaf` и являются тривиальными подклассами классов `Font` и `Color`, реализующими интерфейс-метку `UIResource`. Этот интерфейс позволяет компонентам отличать значения свойств, заданных стилем (все они реализуют `UIResource`), и значения свойств, заданных приложением. Основываясь на этом различении, установки приложения могут замещать установки стиля, даже если стиль (или тема) изменяются в процессе работы приложения.

Пример 10.28. ThemeManager.java

```
package com.davidflanagan.examples.gui;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.plaf.*;
import javax.swing.plaf.metal.MetalLookAndFeel;
import javax.swing.plaf.metal.DefaultMetalTheme;

/**
 * Этот класс считывает описания тем из GUIResourceBundle и использует
 * их для задания шрифтов и цветов для стиля Metal.
 **/
public class ThemeManager {
 JFrame frame;                // Окно, к которому будут применяться темы
 GUIResourceBundle resources; // Описывающие темы свойства

 /**
  * Строим ThemeManager для окна и набора ресурсов. Если задана
  * принимаемая по умолчанию тема, она применяется к окну
  **/
 public ThemeManager(JFrame frame, GUIResourceBundle resources) {
  this.frame = frame;
  this.resources = resources;
  String defaultName = getDefaultThemeName();
  if (defaultName != null) setTheme(defaultName);
 }

 /** Ищем тему, заданную по имени, и применяем ее к окну */
 public void setTheme(String themeName) {
  // Ищем тему в наборе ресурсов
  Theme theme = new Theme(resources, themeName);
  // Назначаем ее текущей темой
  MetalLookAndFeel.setCurrentTheme(theme);
  // Заново применяем стиль Metal, чтобы установить новую тему
  try { UIManager.setLookAndFeel(new MetalLookAndFeel()); }
  catch(UnsupportedLookAndFeelException e) {}
  // Распространяем новый стиль по всему дереву компонентов окна
  SwingUtilities.updateComponentTreeUI(frame);
 }

 /** Получаем "отображаемое название" или метку темы, заданной по имени */
 public String getDisplayName(String themeName) {
  return resources.getString(themeName + ".name", null);
 }

 /** Получаем имя принимаемой по умолчанию темы, или null */
 public String getDefaultThemeName() {
  return resources.getString("defaultTheme", null);
 }

 /**
  * Получаем список всех известных имен тем. Возвращаемые значения являются
```

```
 * именами тем в файле свойств, а не отображаемыми названиями тем.
 **/
public String[] getAllThemeNames() {
 java.util.List names = resources.getStringList("themelist");
 return (String[]) names.toArray(new String[names.size()]);
}

/**
 * Получаем компонент Jmenu, в котором перечислены все известные темы по их
 * отображаемым названиям, и устанавливаем выбранную тему.
 **/
public JMenu getThemeMenu() {
 String[] names = getAllThemeNames();
 String defaultName = getDefaultThemeName();
 JMenu menu = new JMenu("Themes");
 ButtonGroup buttongroup = new ButtonGroup();
 for(int i = 0; i < names.length; i++) {
  final String themeName = names[i];
  String displayName = getDisplayName(themeName);
  JMenuItem item = menu.add(new JRadioButtonMenuItem(displayName));
  buttongroup.add(item);
  if (themeName.equals(defaultName)) item.setSelected(true);
  item.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
     setTheme(themeName);
    }
   });
 }
 return menu;
}

/**
 * Этот класс является подклассом DefaultMetalTheme, который должен
 * возвращать значения Color и Font, считанные из GUIResourceBundle
 **/
public static class Theme extends DefaultMetalTheme {
 // Эти поля являются значениями, возвращаемыми данной темой
 String displayName;
 FontUIResource controlFont, menuFont, smallFont;
 FontUIResource systemFont, userFont, titleFont;
 ColorUIResource primary1, primary2, primary3;
 ColorUIResource secondary1, secondary2, secondary3;

 /**
  * Этот конструктор считывает все необходимые ему значения
  * из GUIResourceBundle. Если свойства не заданы, он делает разумные
  * предположения по умолчанию.
  **/
 public Theme(GUIResourceBundle resources, String name) {
  // Используем этот объект темы в качестве источника принимаемых
  // по умолчанию значений шрифта
  DefaultMetalTheme defaultTheme = new DefaultMetalTheme();
```

```
        // Ищем отображаемое имя темы
        displayName = resources.getString(name + ".name", null);

        // Ищем шрифты для этой темы
        Font control = resources.getFont(name + ".controlFont", null);
        Font menu = resources.getFont(name + ".menuFont", null);
        Font small = resources.getFont(name + ".smallFont", null);
        Font system = resources.getFont(name + ".systemFont", null);
        Font user = resources.getFont(name + ".userFont", null);
        Font title = resources.getFont(name + ".titleFont", null);

        // Преобразуем шрифты к типу FontUIResource или получаем принимаемые
        // по умолчанию
        if (control != null) controlFont = new FontUIResource(control);
        else controlFont = defaultTheme.getControlTextFont();
        if (menu != null) menuFont = new FontUIResource(menu);
        else menuFont = defaultTheme.getMenuTextFont();
        if (small != null) smallFont = new FontUIResource(small);
        else smallFont = defaultTheme.getSubTextFont();
        if (system != null) systemFont = new FontUIResource(system);
        else systemFont = defaultTheme.getSystemTextFont();
        if (user != null) userFont = new FontUIResource(user);
        else userFont = defaultTheme.getUserTextFont();
        if (title != null) titleFont = new FontUIResource(title);
        else titleFont = defaultTheme.getWindowTitleFont();

        // Ищем основной и побочный цвета
        Color primary = resources.getColor(name + ".primary", null);
        Color secondary = resources.getColor(name + ".secondary", null);

        // Производим из этих двух все шесть цветов, при необходимости
        // используя принимаемые по умолчанию
        if (primary != null) primary1 = new ColorUIResource(primary);
        else primary1 = new ColorUIResource(102, 102, 153);
        primary2 = new ColorUIResource(primary1.brighter());
        primary3 = new ColorUIResource(primary2.brighter());
        if (secondary != null) secondary1 = new ColorUIResource(secondary);
        else secondary1 = new ColorUIResource(102, 102, 102);
        secondary2 = new ColorUIResource(secondary1.brighter());
        secondary3 = new ColorUIResource(secondary2.brighter());
    }

    // Эти методы замещают DefaultMetalTheme и возвращают
    // найденные и вычисленные значения для этой темы
    public String getName() { return displayName; }
    public FontUIResource getControlTextFont() { return controlFont;}
    public FontUIResource getSystemTextFont() { return systemFont;}
    public FontUIResource getUserTextFont() { return userFont;}
    public FontUIResource getMenuTextFont() { return menuFont;}
    public FontUIResource getWindowTitleFont() { return titleFont;}
    public FontUIResource getSubTextFont() { return smallFont;}
    protected ColorUIResource getPrimary1() { return primary1; }
    protected ColorUIResource getPrimary2() { return primary2; }
```

```
    protected ColorUIResource getPrimary3() { return primary3; }
    protected ColorUIResource getSecondary1() { return secondary1; }
    protected ColorUIResource getSecondary2() { return secondary2; }
    protected ColorUIResource getSecondary3() { return secondary3; }
  }
}
```

Собственные компоненты

Большая часть примеров этой главы представляет собой подклассы компонентов `JFrame`, `JPanel` или других компонентов Swing. В этом смысле все они были «собственными (custom) компонентами». Класс `ItemChooser` – это особенно полезный пример такого рода компонента. В большинстве, однако, эти примеры не обрабатывали собственных низкоуровневых событий мыши и клавиатуры и не делали свою прорисовку; обработку низкоуровневых событий за них производил родительский класс, а их внешний вид определялся подкомпонентами.

Есть, однако, еще один сорт низкоуровневой настройки компонентов, при котором внешний вид компонента определяется им самим при помощи объекта `Graphics`, а его «чувствительность» – прямой обработкой происходящих в нем событий мыши и клавиатуры. Класс `AppletMenuBar`, приведенный в примере 10.29, – это собственный компонент именно такого типа. Как следует из его имени, этот компонент отображает полосу меню в апплете. Большинство броузеров (во всяком случае, на момент написания этой книги) не содержат встроенных компонентов Swing. Поэтому, чтобы обеспечить совместимость с такими броузерами, многие апплеты пишутся с применением исключительно компонентов AWT. Одним из недостатков AWT является то, что полосы меню могут иметь только объекты `Frame`; класс `Applet` не может отображать `java.awt.Menubar`. Таким образом, компонент `AppletMenuBar` – это собственный компонент AWT, симулирующий полосу меню. Он предоставляет метод `paint()`, рисующий полосу меню, в первую очередь заголовки меню, и определяет методы обработки событий низкого уровня, следящие за мышью, осуществляет эффекты «наезда» («rollover») и, когда нужно, раскрывает меню (с применением класса `java.awt.PopupMenu`). Класс `AppletMenuBar` включает в себя также внутренний класс `Demo`, реализующий демонстрационный апплет.

Чтобы лучше понять компонент `AppletMenuBar`, полезно посмотреть, как он может быть применен в апплете. Ниже показан простой метод `init()` для апплета, использующего этот класс:

```
public void init() {
 AppletMenuBar menubar = new AppletMenuBar(); // Создаем полосу меню
 menubar.setForeground(Color.black);          // Устанавливаем ее свойства
 menubar.setHighlightColor(Color.red);
 menubar.setFont(new Font("helvetica", Font.BOLD, 12));
 this.setLayout(new BorderLayout());
```

```
    this.add(menubar, BorderLayout.NORTH);   // Помещаем ее вверху апплета

    // Создаем пару раскрывающихся меню и вставляем в них элементы-заглушки
    PopupMenu file = new PopupMenu();
    file.add("New..."); file.add("Open..."); file.add("Save As...");
    PopupMenu edit = new PopupMenu();
    edit.add("Cut"); edit.add("Copy"); edit.add("Paste");

    // Помещаем раскрывающиеся меню (с метками) в полосу меню
    menubar.addMenu("File", file);
    menubar.addMenu("Edit", edit);
  }
```

При изучении `AppletMenuBar` следует на многое обратить внимание. Метод `paint()` занимает центральное положение в компоненте, поскольку он его отображает. Метод `processMouseEvent()`, обрабатывающий все события мыши, не менее важен. Оба этих метода замещают методы, унаследованные от класса `Component`, и оба они опираются на метод `measure()`, предварительно вычисляющий размер и положение для каждой надписи. Класс `AppletMenuBar` определяет разнообразные методы доступа к свойствам, и многие методы установки свойств реагируют на изменения свойств, заставляя полосу меню пересчитывать свои размеры и перерисовываться. Другие замещенные методы `getPreferedSize()`, `getMinimumSize()` и `isFocusTraversable()` предоставляют важную информацию о компоненте, используемую контейнером компонента. Заметьте, наконец, что `AppletMenuBar` не выстреливает (fire) никакие события и не определяет никакие методы регистрации слушателей. Объекты `MenuItem` в раскрывающихся меню производят всю обработку событий, так что в этом просто нет никакой нужды. Все же генерирование событий – это самая обычная задача для собственных компонентов, как мы это видели в примере с `ItemChooser`.

По необходимости `AppletMenuBar` является компонентом AWT. Разработка собственных компонентов Swing производится похожим образом, но имеет несколько ключевых отличий. К примеру, компоненты Swing прорисовывают себя при помощи метода `paintComponent()`, а не метода `paint()`. Если вы захотите, чтобы ваш компонент поддерживал подключаемые стили, вам придется определить для него `javax.swing.plaf.ComponentUI`. Дополнительные сведения о собственных компонентах Swing можно почерпнуть в главе 4 книги «Java Foundation Classes in a Nutshell». Наконец, хотя `AppletMenuBar` можно использовать в Swing GUI, этого делать не следует. Во-первых, в этом нет необходимости. Во-вторых, его стиль не соответствует стилю других компонентов Swing. И в-третьих, смешение «тяжеловесных» компонентов, таких как этот, с «легковесными» компонентами Swing может вызвать проблемы при компоновке и перерисовке.

Пример 10.29. AppletMenuBar.java

```
package com.davidflanagan.examples.gui;
import java.awt.*;
```

```
import java.awt.event.*;
import java.util.Vector;

public class AppletMenuBar extends Panel {
 // Содержимое полосы меню
 Vector labels = new Vector();
 Vector menus = new Vector();

 // Свойства
 Insets margins = new Insets(3, 10, 3, 10); // top, left, bottom, right
 int spacing = 10;       // Промежуток между метками меню
 Color highlightColor;  // Цвет метки при помещении на нее указателя мыши

 // Внутренности
 boolean remeasure = true; // Нужно ли пересчитывать размеры надписей
 int[] widths;             // Ширина каждой надписи
 int[] startPositions;     // Начало каждой надписи
 int ascent, descent;      // Размеры шрифтов
 Dimension prefsize = new Dimension(); // Желательные размеры
 int highlightedItem = -1;  // На каком элементе находится указатель мыши?

 /**
 * Создаем новый компонент, симулирующий полосу меню, за счет отображения
 * заданных меток. Когда пользователь щелкает на метке, открывается меню
 * PopupMenu, заданное в массиве меню. Элементом массива меню может быть объект
 * PopupMenu или объект PopupMenuFactory, динамически создающий меню. Возможно,
 * мы также предоставим какой-то другой метод конструктора или фабрики
 * (factory), считывающий раскрывающиеся меню из конфигурационного файла.
 */
 public AppletMenuBar() {
 // Мы будем рады видеть эти события
 enableEvents(AWTEvent.MOUSE_EVENT_MASK |
   AWTEvent.MOUSE_MOTION_EVENT_MASK);
 }

 /** Помещаем в полосу раскрывающееся меню */
 public void addMenu(String label, PopupMenu menu) {
 insertMenu(label, menu, -1);
 }

 /** Вставляем в полосу раскрывающееся меню */
 public void insertMenu(String label, PopupMenu menu, int index) {
 if (index < 0) index += labels.size()+1; // Место вставки
 this.add(menu);                          // Теперь меню наше
 labels.insertElementAt(label, index);    // Запоминаем метку
 menus.insertElementAt(menu, index);      // Запоминаем меню
 remeasure = true;                        // Пересчитываем размеры
 invalidate();                          // Контейнер должен перекомпоноваться
 }

 /** Методы доступа к свойству margins */
 public Insets getMargins() { return (Insets) margins.clone(); }
 public void setMargins(Insets margins) {
 this.margins = margins;
```

```
remeasure = true;
invalidate();
}

/** Методы доступа к свойству spacing */
public int getSpacing() { return spacing; }
public void setSpacing(int spacing) {
if (this.spacing != spacing) {
 this.spacing = spacing;
 remeasure = true;
 invalidate();
}
}

/** Методы доступа к свойству highlightColor */
public Color getHighlightColor() {
if (highlightColor == null) return getForeground();
else return highlightColor;
}
public void setHighlightColor(Color c) {
if (highlightColor != c) {
 highlightColor = c;
 repaint();
}
}

/** Мы замещаем метод setFont(), так чтобы пересчитывались размеры */
public void setFont(Font f) {
super.setFont(f);
remeasure = true;
invalidate();
}

/** Замещаем этот метод установки цвета, так чтобы
    все сразу перерисовывалось новым цветом */
public void setForeground(Color c) {
super.setForeground(c);
repaint();
}
public void setBackground(Color c) {
super.setBackground(c);
repaint();
}

/**
* Этот метод вызывается для перерисовывания компонента.
* Если бы мы реализовали компонент Swing, нужно было бы
* замещать метод paintComponent()
**/
public void paint(Graphics g) {
if (remeasure) measure(); // При необходимости первым делом
                          // пересчитываем размеры

// Вычисляем координату Y рисунка
```

```
Dimension size = getSize();
int baseline = size.height - margins.bottom - descent;
// Устанавливаем шрифт
g.setFont(getFont());
// Цикл по меткам
int nummenus = labels.size();
for(int i = 0; i < nummenus; i++) {
 // Устанавливаем цвет рисования. Выделяем текущий элемент
 if ((i == highlightedItem) && (highlightColor != null))
 g.setColor(getHighlightColor());
 else
 g.setColor(getForeground());

 // Рисуем метку меню в месте, вычисленном методом measure()
 g.drawString((String)labels.elementAt(i),
   startPositions[i], baseline);
}

// Теперь подчеркиваем меню "углубленной" линией.
Color bg = getBackground();
g.setColor(bg.darker());
g.drawLine(0, size.height-2, size.width, size.height-2);
g.setColor(bg.brighter());
g.drawLine(0, size.height-1, size.width, size.height-1);
}

/** Вызывается, когда в полосе меню случается событие мыши */
protected void processMouseEvent(MouseEvent e) {
int type = e.getID();            // Событие какого типа?
int item = findItemAt(e.getX()); // Над какой меткой?

if (type == MouseEvent.MOUSE_PRESSED) {
 // Если это было нажатие кнопки, раскрываем меню
 if (item == -1) return;
 Dimension size = getSize();
 PopupMenu pm = (PopupMenu) menus.elementAt(item);
 if (pm != null) pm.show(this, startPositions[item]-3, size.height);

}
else if (type == MouseEvent.MOUSE_EXITED) {
 // Если указатель мыши покинул полосу меню, снимаем выделение цветом
 if (highlightedItem != -1) {
 highlightedItem = -1;
 if (highlightColor != null) repaint();
 }
}
else if ((type == MouseEvent.MOUSE_MOVED) ||
  (type == MouseEvent.MOUSE_ENTERED)) {
 // Если мышь передвинулась, при необходимости меняем выделенную метку
 if (item != highlightedItem) {
 highlightedItem = item;
 if (highlightColor != null) repaint();
 }
```

```
}
}

/** Этот метод вызывается при перемещениях мыши */
protected void processMouseMotionEvent(MouseEvent e) {
processMouseEvent(e);
}

/** Этот служебный метод преобразует X-координату в индекс метки меню */
protected int findItemAt(int x) {
// поиск мог бы быть и более эффективным...
int nummenus = labels.size();
int halfspace = spacing/2-1;
int i;
for(i = nummenus-1; i >= 0; i--) {
 if ((x >= startPositions[i]-halfspace) &&
 (x <= startPositions[i]+widths[i]+halfspace)) break;
}
return i;
}

/**
* Этот метод "измеряет" метки и вычисляет их положения, что позволяет нам
* узнавать, где произошел щелчок мыши, и правильно перерисовывать полосу меню.
**/
protected void measure() {
// Получаем информацию о шрифте
FontMetrics fm = this.getFontMetrics(getFont());
// Запоминаем основной размер шрифта
ascent = fm.getAscent();
descent = fm.getDescent();
// Создаем массивы для хранения размеров и положений
int nummenus = labels.size();
widths = new int[nummenus];
startPositions = new int[nummenus];

// Измеряем строки меток и вычисляем положение начала каждой из них
int pos = margins.left;
for(int i = 0; i < nummenus; i++) {
 startPositions[i] = pos;
 String label = (String)labels.elementAt(i);
 widths[i] = fm.stringWidth(label);
 pos += widths[i] + spacing;
}

// На основе этих данных вычисляем предпочтительный размер
prefsize.width = pos - spacing + margins.right;
prefsize.height = ascent + descent + margins.top + margins.bottom;

// Теперь больше нечего пересчитывать.
remeasure = false;
}
```

```
    /**
    * Эти методы сообщают контейнеру предпочтительный размер полосы меню.
    *
    **/
    public Dimension getMinimumSize() { return getPreferredSize(); }
    public Dimension getPreferredSize() {
    if (remeasure) measure();
    return prefsize;
    }
    /** @deprecated помещен здесь для совместимости с Java 1.0 */
    public Dimension minimumSize() { return getPreferredSize(); }
    /** @deprecated помещен здесь для совместимости с Java 1.0 */
    public Dimension preferredSize() { return getPreferredSize(); }

    /**
    * Этот метод вызывается при создании основного компонента AWT.
    * Нельзя рассчитать собственные размеры (нет размеров шрифтов)
    * до вызова этого метода.
    **/
    public void addNotify() {
    super.addNotify();
    measure();
    }

    /** Этот метод приказывает контейнеру не давать нам фокус ввода */
    public boolean isFocusTraversable() { return false; }
}
```

Упражнения

10-1. Посмотрите еще раз на рис. 10.7. Напишите класс, производящий образец компоновки, подобный `GridBagLayoutPane`, но без использования менеджера компоновки `GridBagLayout`. Вам, скорее всего, придется использовать менеджер `BorderLayout` и контейнер `Box` или менеджер компоновки `BoxLayout`.

10-2. Класс `ScribblePane2` имеет серьезный недостаток: рисунок стирается, когда его что-то заслоняет, а потом снова открывается взгляду. Переделайте `ScribblePane2` так, чтобы он запоминал каракули пользователя. Вам придется переделать метод `lineto()` так, чтобы он не только рисовал линии, но и сохранял координаты и цвет линии (возможно, с применением объектов `Vector` или `ArrayList`). Объект `JComponent` вызывает свой метод `paintComponent()` всякий раз, когда компонент нужно перерисовать. Прочитайте документацию по этому методу, а затем замените его так, чтобы он перерисовывал панель с использованием сохраненных координат. В заключение для работы с этой новой системой вам понадобится переделать и метод `clear()`.

10-3. Класс `ItemChooser` позволяет задавать элементы только при создании компонента. Добавьте в него методы, позволяющие добавлять и удалять элементы. Пусть эти методы работают вне зависимости от используемого типа представления.

10-4. Приложение `Scribble` определяет и использует класс `ColorAction`, чтобы позволить пользователю выбрать цвет рисования. Добавьте еще класс `LineWidthAction`, предоставляющий возможность выбора толщины рисуемой линии. Включите в компонент `ScribblePane2` метод `setLineWidth()` для рисования толстых линий. Информацию о рисовании толстыми линиями см. в `java.awt.BasicStroke`.

10-5. Один из недостатков класса `FontChooser` состоит в том, что при вызове метода `setSelectedFont()` компоненты `ItemChooser` не обновляются в соответствии с этим новым шрифтом. Модифицируйте `FontChooser` так, чтобы он обновлялся в соответствии со сделанным выбором. Вы, вероятно, захотите переделать `ItemChooser` так, чтобы в дополнение к его методу `setSelectedIndex()` у него были методы `setSelectedLabel()` и `setSelectedValue()`, задающие имя и значение выбранного элемента.

10-6. Переделайте программу `ShowComponent`, добавив в меню команду **Show Properties**. При выборе пользователем этого элемента меню программа должна использовать компонент `PropertyTable` (в отдельном окне) для отображения списка свойств, определенных отображаемым в данный момент компонентом. Чтобы сделать это возможным, вам придется следить за отображаемыми компонентами и понадобится опросить `JtabbedPane`, чтобы узнать, какой компонент отображается в данный момент.

Сумев правильно отобразить таблицу свойств, добавьте элемент меню в окно отображения свойств. Будучи выбранным, этот элемент должен находить выбранную строку таблицы (используйте `getSelectedRow()`), а затем позволять пользователю устанавливать значение соответствующего свойства. Используйте один из статических методов `JOptionPane` для отображения диалога, позволяющего пользователю вводить значение свойства (как строку), затем определите по этой строке значение и установите заданное свойство.

10-7. Снова переделайте программу `ShowComponent` так, чтобы в ней появился элемент меню **Containment Hierarchy**. Когда пользователь выберет этот элемент, программа должна использовать компонент `ComponentTree` (в отдельном окне) для отображения иерархии контейнеров отображаемого в данный момент компонента.

10-8. Как правило, у броузеров есть меню **Go**, в котором перечислены 10–15 URL, посещенных последними. Это меню дает быстрый способ повторного посещения этих сайтов. Добавьте эту возможность в класс `WebBrowser`. Обратите внимание, что `WebBrowser` уже

отслеживает историю своих просмотров. Меню **Go** нужно поместить в компонент `JmenuBar`, считанный из `GUIResourceBundle`. Содержимое меню **Go** зависит от текущей истории просмотров. Можно менять содержимое меню всякий раз, когда пользователь посещает очередную страницу. Либо можно использовать объект `MenuListener`, чтобы получать уведомление непосредственно перед тем, как меню откроется, и тогда при помощи метода `menuSelected()` слушателя помещать в меню соответствующие элементы.

10-9. Класс `GUIResourceBundle` и интерфейс расширения `ResourceParser` предоставляют мощный механизм описания GUI в файле свойств. Реализуйте другие классы `ResourceParser`, поддерживающие ресурсы других типов. В частности, напишите анализаторы для классов `ImageIcon` и `KeyStroke`; поддержка этих двух классов необходима для поддержки класса `ActionParser`.

10-10. Переделайте класс `ShowComponent`, чтобы он использовал класс `ThemeManager` (с подходящим файлом свойств) и меню **Themes**, которое он может создать. В идеале меню **Themes** следовало бы организовать как подменю меню **Look and Feel**, уже отображаемого программой.

10-11. Преобразуйте компонент `AppletMenuBar` так, чтобы в дополнение к отображению меню он мог отображать кнопки (то есть он должен работать как комбинация полосы меню и панели инструментов). Добавьте метод `addCommand()`, аналогичный методу `addMenu()`. Вместо строки и объекта `PopupMenu` этот новый метод должен принимать в качестве аргументов строку метки и объект `ActionListener`. Когда пользователь щелкает на заданной метке, компонент должен вызывать метод `actionPerformed()` заданного слушателя. При тестировании этого класса имейте в виду, что класс `Command` из главы 8 реализует интерфейс `ActionListener`.

Глава 11

Графика

Версии Java 1.0 и 1.1 предоставляли рудиментарные графические возможности при посредстве класса `java.awt.Graphics` и связанных с ним классов, таких как `java.awt.Color` и `java.awt.Font`. Начиная с Java 1.2 Java 2D API предоставляет современные графические средства двумерного рисования с использованием класса `java.awt.Graphics2D` (подкласс `Graphics`) и связанных с ним классов, таких как `java.awt.BasicStroke`, `java.awt.GradientPaint`, `java.awt.TexturePaint`, `java.awt.AffineTransform` и `java.awt.AlphaComposite`.

В этой главе демонстрируется применение этих классов; в ней показано, как можно создавать рисунки с применением Java 2D API и без него. Ключевым классом при выполнении всех графических операций является `Graphics` (или, в Java 2D API, его подкласс `Graphics2D`). Назначение этого класса состоит в реализации трех функций:

Он определяет поверхность рисования (drawing surface)

Объект `Graphics` может представлять экранную область рисования внутри компонента AWT (или Swing). Он может представлять также внеэкранное изображение, на котором можно рисовать, или лист бумаги в принтере.

Он определяет методы рисования

Все примитивные графические операции, такие как рисование линий, заливка (цветом) фигур и вывод текста, выполняются с применением методов класса `Graphics`.

Он определяет атрибуты, используемые методами рисования

Различные методы класса `Graphics` могут устанавливать шрифт, цвет, область отсечки и другие атрибуты, используемые при выполнении графических операций, таких как рисование линий, заливка

фигур (цветом) и вывод текста. Значения этих графических атрибутов часто являются экземплярами классов AWT, таких как `Color`, `Font`, `Stroke`, `AffineTransform` и т. д.

Графические способности Java тесно связаны с Abstract Windowing Toolkit (AWT). Собственно классы `Graphics` и `Graphics2D` являются частью пакета `java.awt`, как и все связанные с ними классы, определяющие графические атрибуты. Как мы уже обсуждали в главе 10 «Графические интерфейсы пользователя (GUI)», центральным классом AWT является `java.awt.Component`. Класс `Component` – это краеугольный камень всех графических пользовательских интерфейсов Java. Объект `Graphics` представляет поверхность рисования, но язык Java не позволяет рисовать прямо на экране компьютера. Вместо этого он предписывает рисовать только в пределах компонента (`Component`), используя объект `Graphics`, представляющий поверхность рисования этого компонента. Таким образом, чтобы заниматься графикой в Java, вам нужно иметь компонент `Component` (или апплет `java.applet.Applet`, подкласс `Component`), в котором можно рисовать. Рисование осуществляется обычно путем создания подкласса определенного компонента и определения для него метода `paint()` (или метода `paintComponent()` для компонентов Swing). Примеры в этой главе построены так, чтобы основное внимание уделить приемам рисования, а не приемам построения GUI, но здесь можно видеть и код, выполняющий задачи, связанные с GUI.

Графика до Java 1.2

До появления Java 2D API в Java 1.2 класс `java.awt.Graphics` предоставлял только рудиментарные графические возможности. Он позволял рисовать линии, рисовать и заливать цветом простые фигуры, отображать текст и изображения и осуществлять основные манипуляции с изображениями. Можно было задавать цвет, шрифт, область отсечки и положение текущей точки. Особенно не хватало, однако, возможности задавать толщину линии, поворачивать рисунок и масштабировать его. Пример 11.1 – это листинг апплета `GraphicsSampler`, демонстрирующий имевшиеся до Java 1.2 базовые графические примитивы и атрибуты. Вывод этого апплета показан на рис. 11.1.

Если вы уже читали главу 15 «Апплеты», то должны разобраться в структуре этого кода. Апплет, однако, очень прост, так что в нем можно разобраться, и не читав этой главы. Метод `init()` вызывается один раз, когда апплет запускается впервые, и выполняет начальную (one-time) инициализацию. Метод `paint()` отображает содержимое апплета. Объект `Graphics`, передаваемый этому методу, представляет поверхность рисования в апплете; он позволяет рисовать в любом месте в пределах объекта `Applet`. Служебные методы `tile()` и `centerText()` определены в помощь методу `paint()`. Они демонстрируют важные приемы рисования, но не имеют отношения к API апплета.

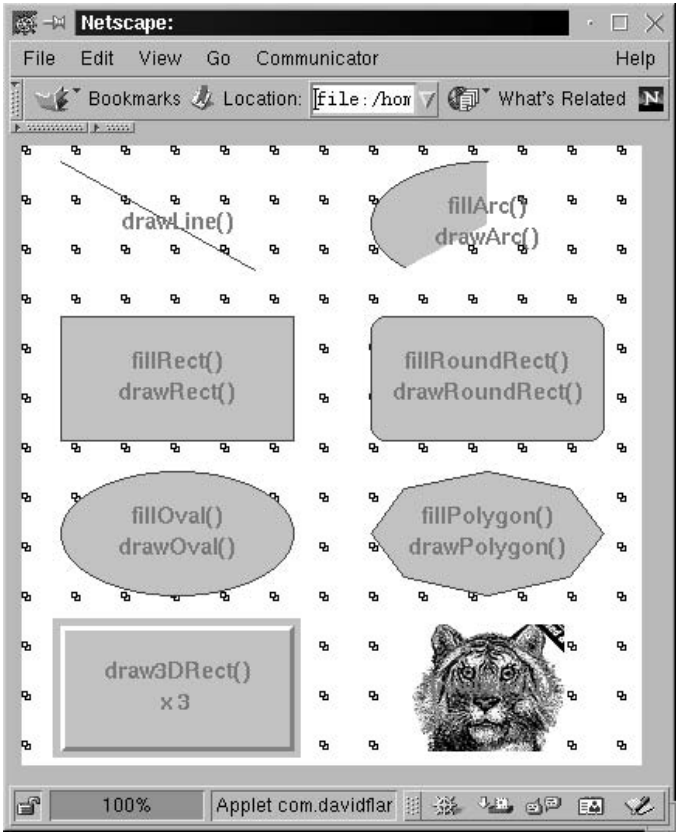


Рис. 11.1. *Образцы графических примитивов*

Важно понимать, что метод `paint()` может вызываться неоднократно. Рисунок, выполненный этим методом, не пребывает неизменным: если накрыть апплет другим окном, а затем окно убрать, содержащаяся в апплете графика может быть утрачена (произойдет ли это на самом деле, зависит от используемой операционной системы и других факторов). Чтобы восстановить рисунок, система снова вызывает метод `paint()`. Таким образом, все рисование должно выполняться методом `paint()`. Если что-то нарисовано методом `init()`, эта графика может быть стерта и может никогда не появиться вновь. (Это один из серьезных недостатков примера 10.11 и связанных с ним примеров из главы 10.)

Пример 11.1. GraphicsSampler.java

```
package com.davidflanagan.examples.graphics;
import java.applet.*;
import java.awt.*;
```

```
/**
 * Апплет, демонстрирующий большинство графических примитивов из
 * java.awt.Graphics.
 **/
public class GraphicsSampler extends Applet {
    Color fill, outline, textcolor;  // Разные используемые цвета
    Font font;                       // Шрифт, используемый для текста
    FontMetrics metrics;             // Информация о размере
    Image image, background;         // Изображения, при помощи которых
                                     // мы будем рисовать

    // Этот метод вызывается, когда апплет создается впервые.
    // Он производит инициализацию, такую как создание ресурсов
    // (значений графических атрибутов), используемых методом paint().
    public void init() {
        // Инициализируем цветовые ресурсы. Обратите внимание на конструктор
        // Color() и применение заранее определенных цветовых констант.
        fill = new Color(200, 200, 200); // Красный, синий и зеленый равной
                                         // интенсивности == серый
        outline = Color.blue;            // То же, что new Color(0, 0, 255)
        textcolor = Color.red;           // То же, что new Color(255, 0, 0)

        // Создаем шрифт, который будет использоваться методом paint().
        // Также получаем его размеры.
        font = new Font("sansserif", Font.BOLD, 14);
        metrics = this.getFontMetrics(font);

        // Загружаем объекты Image для использования в методе paint().
        image = this.getImage(this.getDocumentBase(), "tiger.gif");
       background = this.getImage(this.getDocumentBase(), "background.gif");

        // Устанавливаем свойство, задающее фоновый цвет для апплета
        this.setBackground(Color.lightGray);
    }

    // Этот метод вызывается всякий раз, когда апплет рисуется
    // или перерисовывается
    public void paint(Graphics g) {
        g.setFont(font);  // Задаем используемый шрифт

        // Рисуем фон, размножая изображение при помощи метода tile(),
        // определенного ниже
        tile(g, this, background);

        // Проводим линию
        g.setColor(outline);            // Задаем цвет линии
        g.drawLine(25, 10, 150, 80);    // Проводим линию от (25,10)
                                        // до (150,80)
        // Выводим текст.  См. ниже определение метода centerText().
        centerText("drawLine()", null, g, textcolor, 25, 10, 150, 80);

        // Рисуем и заливаем дугу
        g.setColor(fill);
        g.fillArc(225, 10, 150, 80, 90, 135);
```

```
g.setColor(outline);
g.drawArc(225, 10, 150, 80, 90, 135);
centerText("fillArc()", "drawArc()", g, textcolor,225,10,150,80);

// Рисуем и заливаем прямоугольник
g.setColor(fill);
g.fillRect(25, 110, 150, 80);
g.setColor(outline);
g.drawRect(25, 110, 150, 80);
centerText("fillRect()", "drawRect()", g, textcolor, 25, 110, 150, 80);

// Рисуем и заливаем прямоугольник со скругленными углами
g.setColor(fill);
g.fillRoundRect(225, 110, 150, 80, 20, 20);
g.setColor(outline);
g.drawRoundRect(225, 110, 150, 80, 20, 20);
centerText("fillRoundRect()", "drawRoundRect()", g, textcolor,
           225, 110, 150, 80);

// Рисуем и заливаем овал
g.setColor(fill);
g.fillOval(25, 210, 150, 80);
g.setColor(outline);
g.drawOval(25, 210, 150, 80);
centerText("fillOval()", "drawOval()", g, textcolor, 25, 210, 150, 80);

// Рисуем восьмиугольник, используя массивы координат X и Y
int numpoints = 8;
int[] xpoints = new int[numpoints+1];
int[] ypoints = new int[numpoints+1];
for(int i=0; i < numpoints; i++) {
    double angle = 2*Math.PI * i / numpoints;
    xpoints[i] = (int)(300 + 75*Math.cos(angle));
    ypoints[i] = (int)(250 - 40*Math.sin(angle));
}

// Рисуем и заливаем многоугольник
g.setColor(fill);
g.fillPolygon(xpoints, ypoints, numpoints);
g.setColor(outline);
g.drawPolygon(xpoints, ypoints, numpoints);
centerText("fillPolygon()", "drawPolygon()", g, textcolor,
           225, 210, 150, 80);

// Рисуем "трехмерный" прямоугольник (сначала очищаем область для него)
g.setColor(fill);
g.fillRect(20, 305, 160, 90);
g.draw3DRect(25, 310, 150, 80, true);
g.draw3DRect(26, 311, 148, 78, true);
g.draw3DRect(27, 312, 146, 76, true);
centerText("draw3DRect()", "x 3", g, textcolor, 25, 310, 150, 80);

// Рисуем изображение (по центру области рисования)
int w = image.getWidth(this);
int h = image.getHeight(this);
```

```
        g.drawImage(image, 225 + (150-w)/2, 310 + (80-h)/2, this);
        centerText("drawImage()", null, g, textcolor,  225, 310, 150, 80);
    }

    // Служебный метод для укладки (tile) изображения по фону компонента
    protected void tile(Graphics g, Component c, Image i) {
        // Для совместимости с Java 1.0 и старыми броузерами типа Netscape 3
        // используйте bounds() вместо getBounds()
        Rectangle r = c.getBounds();              // Размеры компонента?
        int iw = i.getWidth(c);                   // Размеры изображения?
        int ih = i.getHeight(c);
        if ((iw <= 0) || (ih <= 0)) return;
        for(int x=0; x < r.width; x += iw)        // Цикл по горизонтали
            for(int y=0; y < r.height; y += ih)   // Цикл по вертикали
                g.drawImage(i, x, y, c);          // Выводим изображение
    }

    // Служебный метод, центрирующий две строки текста в прямоугольнике.
    // Основывается на значении FontMetrics, полученном в методе init().
    protected void centerText(String s1, String s2, Graphics g, Color c,
                              int x, int y, int w, int h)
    {
        int height = metrics.getHeight();  // Высота шрифта?
        int ascent = metrics.getAscent();  // Где опорная линия?
        int width1=0, width2 = 0, x0=0, x1=0, y0=0, y1=0;
        width1 = metrics.stringWidth(s1);  // Какой ширины строки?
        if (s2 != null) width2 = metrics.stringWidth(s2);
        x0 = x + (w - width1)/2;          // Центрируем строки по горизонтали
        x1 = x + (w - width2)/2;
       if (s2 == null)                   // Центрируем одну строку по вертикали
            y0 = y + (h - height)/2 + ascent;
        else {                            // Центрируем две строки по вертикали
            y0 = y + (h - (int)(height * 2.2))/2 + ascent;
            y1 = y0 + (int)(height * 1.2);
        }
        g.setColor(c);                    // Устанавливаем цвет
        g.drawString(s1, x0, y0);         // Выводим строки
        if (s2 != null) g.drawString(s2, x1, y1);
    }
}
```

Запуск апплета

Если вы еще не читали главу 15, то, возможно, не знаете, как надо запускать апплет из примера 11.1. Все, что для этого нужно сделать, – это включить тег `<applet>` в HTML-файл, поместить файл в тот же каталог, где находится файл класса `GraphicsSampler`, и загрузить этот HTML-файл в броузер:

```
<applet code="com/davidflanagan/examples/graphics/GraphicsSampler.class"
        codebase="../../../.."
```

```
        width=400 height=400>
</applet>
```

Этот тег можно найти в файле *GraphicsSampler.html* в онлайновом архиве примеров к этой книге.

Обратите внимание на то, что `GraphicsSampler` вызывает метод `getBounds()`, принадлежащий Java 1.1, поэтому нужно исспользовать броузер, поддерживающий Java 1.1, такой как Netscape Navigator 4.0 или более поздних версий. Можно также применить приложение *appletviewer*, поставляемое вместе с Java SDK:

```
% appletviewer GraphicsSampler.html
```

Шрифты

Для улучшения межплатформенной совместимости Java 1.0 и 1.1 определяют только минимум шрифтов (как мы увидим ниже, Java 1.2 позволяет использовать все шрифты, имеющиеся в операционной системе). В Java 1.0 это были шрифты TimesRoman, Helvetica и Courier. В Java 1.1 шрифтам были даны предпочтительные логические имена: Serif, SansSerif и Monospaced. Для каждого шрифта доступны четыре начертания и любое число размеров. Пример 11.2 представляет листинг апплета `FontList`, отображающего все доступные шрифты во всех доступных начертаниях. Его вывод показан на рис. 11.2.

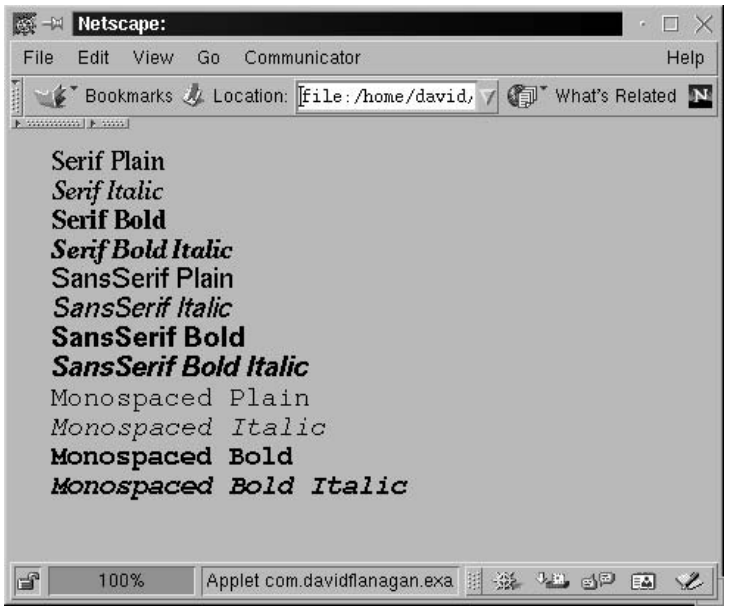


***Рис. 11.2.** Стандартные шрифты и начертания*

Пример 11.2. FontList.java

```
package com.davidflanagan.examples.graphics;
import java.applet.*;
import java.awt.*;

/**
 * Апплет, отображающий стандартные шрифты и начертания, доступные в Java 1.1
 **/
public class FontList extends Applet {
  // Доступные гарнитуры шрифтов
  String[] families = {"Serif",         // "TimesRoman" в Java 1.0
                       "SansSerif",     // "Helvetica" в Java 1.0
                       "Monospaced"};   // "Courier" в Java 1.0

  // Доступные начертания и их имена
  int[] styles = {Font.PLAIN, Font.ITALIC, Font.BOLD, Font.ITALIC+Font.BOLD};
  String[] stylenames = {"Plain", "Italic", "Bold", "Bold Italic"};

  // Рисуем апплет
  public void paint(Graphics g) {
    for(int f=0; f < families.length; f++) {          // Для каждой гарнитуры
      for(int s = 0; s < styles.length; s++) {       // Для каждого начертания
        Font font = new Font(families[f],styles[s],18);  // Создаем шрифт
        g.setFont(font);                                  // Устанавливаем шрифт
        String name = families[f] + " " +stylenames[s];  // Создаем имя
        g.drawString(name, 20, (f*4 + s + 1) * 20);      // Отображаем имя
      }
    }
  }
}
```

Цвета

В Java цвет представляется классом `java.awt.Color`. Класс `Color` определяет множество констант, ссылающихся на предопределенные объекты `Color` для часто используемых цветов, такие как `Color.black` и `Color.white`. Можно также создавать собственные (custom) объекты `Color`, задавая красную, синюю и зеленую составляющие цвета. Эти составляющие можно задавать целыми числами в промежутке от 0 до 255 или как значения типа `float` в промежутке между 0.0 и 1.0. Кроме того, статический метод `getHSBColor()` позволяет создать объект `Color` на основе значений цветового тона (hue), насыщенности (saturation) и яркости (brightness). Начиная с Java 1.1 подкласс `java.awt.SystemColor` класса `Color` определяет множество константных объектов `SystemColor`, представляющих стандартные цвета системной палитры рабочего стола. Вы можете использовать эти цвета, чтобы сделать ваше приложение соответствующим системной схеме цветов.

Пример 11.3 представляет апплет `ColorGradient`, использующий объекты `Color` для создания цветового градиента, показанного на рис. 11.3. Этот апплет использует для определения начального и конечного цве-

Рис. 11.3. *Цветовой градиент*

тов градиента параметры, заданные в файле HTML. Можно запустить его с такими, например, тегами HTML:

```
<applet code="com/davidflanagan/examples/graphics/ColorGradient.class"
    codebase="../../../../" width=525 height=150>
  <param name="startColor" value="#a06060"> <!-- светло-красный -->
  <param name="endColor" value="#ffffff">   <!-- белый -->
</applet>
```

Пример 11.3. ColorGradient.java

```
package com.davidflanagan.examples.graphics;
import java.applet.*;
import java.awt.*;

/** Апплет, демонстрирующий класс Color */
public class ColorGradient extends Applet {
 Color startColor, endColor; // Начальный и конечный цвета градиента
 Font bigFont;               // Используемый шрифт

 /**
  * Получаем начальный и конечный цвета градиента, заданные как параметры
  * апплета, и разбираем их при помощи Color.decode(). Если они заданы
  * в неверном формате, используется белый цвет.
  **/
 public void init() {
  try {
   startColor = Color.decode(getParameter("startColor"));
   endColor = Color.decode(getParameter("endColor"));
  }
  catch (NumberFormatException e) {
   startColor = endColor = Color.white;
  }
```

```
 bigFont = new Font("Helvetica", Font.BOLD, 72);
}

/** Рисуем апплет. Интересный код содержится ниже, в fillGradient()*/
public void paint(Graphics g) {
 fillGradient(this, g, startColor, endColor); // Отображаем градиент
 g.setFont(bigFont);                          // Устанавливаем шрифт
 g.setColor(new Color(100, 100, 200));        // Голубой
 g.drawString("Colors!", 100, 100);           // Рисуем кое-что интересное
}

/**
 * Изображаем цветовой градиент от верхнего края заданного компонента до его
 * нижнего края. Начинаем с начального цвета и плавно переходим к конечному
 **/
public void fillGradient(Component c, Graphics g, Color start, Color end) {
 Rectangle bounds = this.getBounds(); // Каков размер компонента?
 // Получаем красную, синюю и зеленую составляющие начального и конечного
 // цветов как числа с плавающей точкой от 0.0 до 1.0. Обратите внимание
 // на то, что класс Color работает также с целыми значениями от 0 до 255
 float r1 = start.getRed()/255.0f;
 float g1 = start.getGreen()/255.0f;
 float b1 = start.getBlue()/255.0f;
 float r2 = end.getRed()/255.0f;
 float g2 = end.getGreen()/255.0f;
 float b2 = end.getBlue()/255.0f;
 // Вычисляем, насколько должна меняться каждая составляющая
 // при каждом шаге по координате y
 float dr = (r2-r1)/bounds.height;
 float dg = (g2-g1)/bounds.height;
 float db = (b2-b1)/bounds.height;

 // Теперь проходим в цикле каждую строку пикселов компонента
 for(int y = 0; y < bounds.height; y++) {
  g.setColor(new Color(r1, g1, b1));   // Устанавливаем цвет строки
  g.drawLine(0, y, bounds.width-1, y); // Рисуем строку
  r1 += dr; g1 += dg; b1 += db;        // Увеличиваем цветовые составляющие
 }
}
}
```

Простая анимация

Анимация – это не что иное, как быстрое рисование изображений с небольшими изменениями от «кадра» к «кадру», в результате чего создается видимость движения изображения. Пример 11.4 показывает листинг апплета, выполняющего простую анимацию: он перемещает по экрану красный кружок. Апплет использует объект `Thread` для управления моментами перерисовывания. Используемая здесь техника очень проста: запустив апплет вы, возможно, заметите, что он мерцает, когда кружок движется.

Ниже в этой главе я покажу вам другой пример анимации, в котором используются более сложные приемы для устранения мерцания и создания более плавной анимации.

Пример 11.4. BouncingCircle.java

```
package com.davidflanagan.examples.graphics;
import java.applet.*;
import java.awt.*;

/** Апплет, демонстрирующий простую анимацию */
public class BouncingCircle extends Applet implements Runnable {
 int x = 150, y = 50, r = 50; // Положение и радиус кружка
 int dx = 11, dy = 7;         // Траектория кружка
 Thread animator;             // Поток, исполняющий анимацию
 volatile boolean pleaseStop; // Флаг запроса на остановку потока

 /** Этот метод просто рисует кружок в заданном месте */
 public void paint(Graphics g) {
  g.setColor(Color.red);
  g.fillOval(x-r, y-r, r*2, r*2);
 }

 /**
  * Этот метод перемещает кружок (с учетом отскоков), а затем запрашивает
  * перерисовку. Анимационный поток периодически вызывает этот метод.
  **/
 public void animate() {
  // При ударе о край кружок отскакивает.
  Rectangle bounds = getBounds();
  if ((x - r + dx < 0) || (x + r + dx > bounds.width)) dx = -dx;
  if ((y - r + dy < 0) || (y + r + dy > bounds.height)) dy = -dy;

  // Перемещаем кружок.
  x += dx; y += dy;

  // Просим броузер вызвать метод paint() для перерисовывания кружка
  // на новом месте.
  repaint();
 }

 /**
  * Это метод из интерфейса Runnable. Это тело потока исполнения,
  * осуществляющего анимацию. Сам поток создается и запускается
  * методом start().
  **/
 public void run() {
  while(!pleaseStop) {   // Цикл до тех пор, пока не будет
                         // команды остановиться
   animate();            // Обновляем положение и перерисовываем
   try { Thread.sleep(100); }       // Ждем 100 миллисекунд
   catch(InterruptedException e) {} // Игнорируем прерывания
  }
 }
```

```
 /** Запускаем анимацию при запуске апплета броузером */
 public void start() {
  animator = new Thread(this); // Создаем поток исполнения
  pleaseStop = false;          // Пока что не просим его остановиться
  animator.start();            // Запускаем поток.
  // Поток, в котором стартовал этот поток, возвращается к месту вызова.
  // А в это время новый поток animator запускает свой метод run()
 }

 /** Останавливаем анимацию, когда броузер останавливает апплет */
 public void stop() {
  // Устанавливаем флаг, дающий методу run() команду остановиться
  pleaseStop = true;
 }
}
```

Java 2D API

Обратимся теперь к Java 2D API, который становится доступным начиная с Java 1.2. Возможности этого API включают в себя:

- Класс `Graphics2D`, подкласс `Graphics`, определяющий добавочные графические примитивы и атрибуты.
- Интерфейс `Shape`, который Java 2D использует для определения многих графических примитивов и других операций. Пакет `java.awt.geom` содержит множество полезных реализаций `Shape`.
- Интерфейс `Stroke`, описывающий, как линии проводятся (или рисуются, например, широкой кистью). Класс `BasicStroke` реализует этот интерфейс и поддерживает рисование широких и пунктирных линий.
- Интерфейс `Paint`, описывающий заливку фигур. Его реализация `GradientPaint` позволяет произвести заливку цветовым градиентом, а `TexturePaint` поддерживает заливку «плиткой» (tile, фигура заполняется размноженным изображением). Также начиная с Java 1.2 класс `Color` реализует интерфейс `Paint`, который позволяет заливать фигуры сплошным цветом.
- Интерфейс `Composite`, определяющий, как цвета рисунка взаимодействуют с цветами поверхности рисования. Класс `AlphaComposite` позволяет комбинировать цвета на основе уровня прозрачности (alpha transparency). Класс `Color` также содержит новые конструкторы и методы, поддерживающие прозрачные цвета.
- Класс `RenderingHints` позволяющий приложению запрашивать специальные типы рисования, такие как сглаженное (antialiased) рисование. Сглаживание использует прозрачные цвета и композицию цветов для сглаживания краев букв и других фигур, предупреждая возникновение «ступенек» (jaggies).

- Класс `java.awt.geom.AffineTransform`, осуществляющий трансформации систем координат и позволяющий выполнять перемещение, масштабирование, вращение и перекос фигур (и целых координатных систем). Поскольку `AffineTransform` допускает преобразование систем координат, включая масштабирование, Java 2D API проводит различие между системой координат устройства (то есть пространством устройства) и (возможно) трансформированной системой координат пользователя (то есть пространством пользователя). Координаты в пространстве пользователя могут задаваться значениями типа `float` или `double`; графика, выполняемая Java 2D, не зависит от разрешения устройства.

Кроме того, начиная с Java 1.2 ваши приложения уже не ограничены фиксированным набором логических шрифтов; теперь можно применять любой установленный в системе шрифт. Наконец, в Java 2D API включен удобный в использовании API обработки изображений.

Оставшаяся часть этой главы посвящена примерам, демонстрирующим перечисленные возможности Java 2D API. Большинство примеров являются реализациями интерфейса `GraphicsExample`, показанного в примере 11.5.

Пример 11.5. GraphicsExample.java

```
package com.davidflanagan.examples.graphics;
import java.awt.*;

/**
 * Этот интерфейс определяет методы, которые должны быть реализованы объектом,
 * которому надлежит отображаться при помощи объекта GraphicsExampleFrame
 */
public interface GraphicsExample {
 public String getName();       // Возвращает имя графического примера
 public int getWidth();         // Возвращает его ширину
 public int getHeight();        // Возвращает его высоту
 public void draw(Graphics2D g, Component c); // Рисует пример
}
```

Эти примеры можно запускать при помощи программы `GraphicsExampleFrame`, приведенной в конце главы. Она создает простой GUI, создает объекты классов, перечисленных в командной строке, и отображает их в окне. Чтобы, например, отобразить рассматриваемый в следующем разделе пример `Shapes`, можно запустить `GraphicsExampleFrame` так:

```
% java com.davidflanagan.examples.graphics.GraphicsExampleFrame Shapes
```

Многие рисунки этой главы выведены с высоким разрешением; это не просто снимки экранов (screenshot) работающей программы. Эти рисунки – результат фунций печати, включенных в класс `GraphicsExampleFrame`. И хотя сам класс `GraphicsExampleFrame` ничего не рисует, а скорее демонстрирует возможности Java по выводу на печать, он все же приведен в примере 11.18 этой главы.

Рисование и заливка фигур

Пример 11.6 представляет реализацию GraphicsExample, показывающую, как можно создать, нарисовать и закрасить различные объекты Shape. Пример выполняет вывод, показанный на рис. 11.4. Хотя Java 2D API позволяет рисовать и заливать фигуры при помощи методов, продемонстрированных в примере 11.1, в этом примере используется другой подход. Здесь фигуры определяются с помощью различных классов, в основном из пакета java.awt.geom, как объекты Shape.

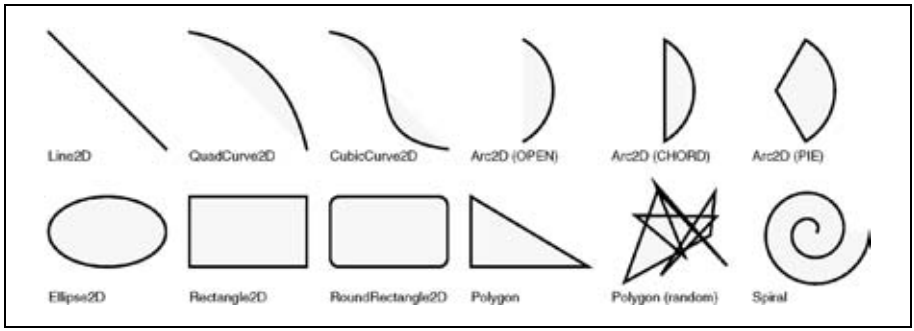


***Рис. 11.4.** Рисование и заливка фигур при помощи Java 2D API*

Каждая фигура Shape рисуется при помощи метода draw() класса Graphics2D и закрашивается при помощи метода fill(). Обратите внимание на то, что каждый объект Shape определен так, что один из его углов расположен в начале координат (или около него), а не в том месте экрана, где он будет отображаться. Созданный таким способом объект не зависит от положения и может использоваться многократно. Чтобы нарисовать фигуру в каком-то конкретном месте, в примере используется метод translate() класса Graphics2D для перемещения начала координат. Наконец, вызов setStroke() указывает, что рисовать нужно линией толщиной в два пиксела, а вызов setRenderingHint() требует, чтобы в процессе рисования использовалось сглаживание (antialiasing).

Пример 11.6. Shapes.java

```
package com.davidflanagan.examples.graphics;
import java.awt.*;
import java.awt.geom.*;
import java.awt.font.*;
import java.awt.image.*;

/** Демонстрация фигур Java2D */
public class Shapes implements GraphicsExample {
 static final int WIDTH = 725, HEIGHT = 250;   // Размер примера
 public String getName() {return "Shapes";}    // Из GraphicsExample
 public int getWidth() { return WIDTH; }       // Из GraphicsExample
 public int getHeight() { return HEIGHT; }     // Из GraphicsExample
```

```
Shape[] shapes = new Shape[] {
// Отрезок прямой линии
new Line2D.Float(0, 0, 100, 100),
// Квадратичная кривая Безье. Две концевые точки и одна управляющая
new QuadCurve2D.Float(0, 0, 80, 15, 100, 100),
// Кубическая кривая Безье. Две концевые точки и две управляющие
new CubicCurve2D.Float(0, 0, 80, 15, 10, 90, 100, 100),
// Часть эллипса в 120 градусов
new Arc2D.Float(-30, 0, 100, 100, 60, -120, Arc2D.OPEN),
// Часть эллипса в 120 градусов, замкнутая хордой
new Arc2D.Float(-30, 0, 100, 100, 60, -120, Arc2D.CHORD),
// Ломтик эллипса в 120 градусов
new Arc2D.Float(-30, 0, 100, 100, 60, -120, Arc2D.PIE),
// Эллипс
new Ellipse2D.Float(0, 20, 100, 60),
// Прямоугольник
new Rectangle2D.Float(0, 20, 100, 60),
// Прямоугольник с закругленными углами
new RoundRectangle2D.Float(0, 20, 100, 60, 15, 15),
// Треугольник
new Polygon(new int[] { 0, 0, 100 }, new int[] {20, 80, 80}, 3),
// Произвольный многоугольник, инициализируемый ниже
null,
// Спираль: экземпляр самостоятельно определяемой реализации Shape
new Spiral(50, 50, 5, 0, 50, 4*Math.PI),
};

{ // Инициализируем вышеупомянутую фигуру null как многоугольник Polygon
     со случайными вершинами
Polygon p = new Polygon();
for(int i = 0; i < 10; i++)
 p.addPoint((int)(100*Math.random()), (int)(100*Math.random()));
shapes[10] = p;
}

// Это подписи под фигурами
String[] labels = new String[] {
"Line2D", "QuadCurve2D", "CubicCurve2D", "Arc2D (OPEN)",
"Arc2D (CHORD)", "Arc2D (PIE)", "Ellipse2D", "Rectangle2D",
"RoundRectangle2D", "Polygon", "Polygon (random)", "Spiral"
};

/** Рисуем пример */
public void draw(Graphics2D g, Component c) {
// Устанавливаем основные атрибуты рисования
g.setFont(new Font("SansSerif", Font.PLAIN, 10)); // выбираем шрифт
g.setStroke(new BasicStroke(2.0f));   // линии толщиной 2 пиксела
g.setRenderingHint(RenderingHints.KEY_ANTIALIASING, // сглаживание
   RenderingHints.VALUE_ANTIALIAS_ON);
g.translate(10, 10);     // поля

// Цикл по фигурам
```

```
 for(int i = 0; i < shapes.length; i++) {
  g.setColor(Color.yellow);        // Устанавливаем цвет
  g.fill(shapes[i]);               // Заливаем им фигуру
  g.setColor(Color.black);         // Переключаемся на черный
  g.draw(shapes[i]);               // Обводим этим цветом фигуру
  g.drawString(labels[i], 0, 110); // Подписываем фигуру
  g.translate(120, 0);             // Перемещаемся к следующей фигуре
  if (i % 6 == 5) g.translate(-6*120, 120); // Нарисовав 6 фигур,
                                            // смещаемся вниз
 }
 }
}
```

Трансформации

Класс `AffineTransform` может трансформировать фигуры (или систему координат), перемещая, масштабируя, вращая и перекашивая их, а также применяя произвольные сочетания всех этих преобразований. Рисунок 11.5 иллюстрирует действие трансформаций координат различных типов на одну фигуру, нарисованную несколько раз. Пример 11.7 показывает код Java, использованный для создания этого рисунка.

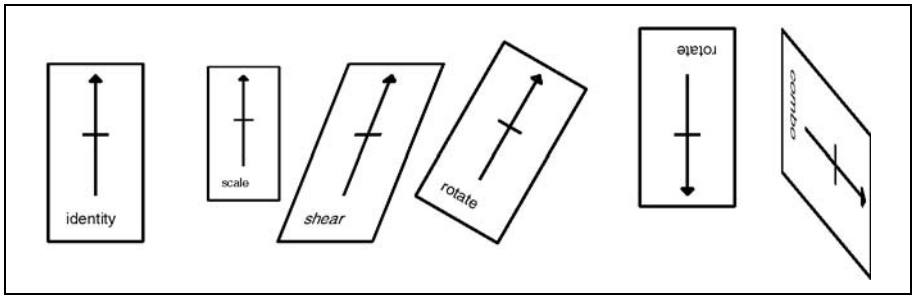


***Рис. 11.5.** Трансформированные фигуры*

Аффинная (affine) *трансформация* – это линейная трансформация, обладающая двумя важными свойствами: все прямые линии остаются прямыми, а все параллельные линии остаются параллельными. Класс `AffineTransform` может выполнить множество самых разных трансформаций, но не может производить нелинейные эффекты, например искажение изображения, которое происходит при рассматривании рисунка через толстую линзу. Эффекты такого рода могут достигаться только при использовании техники обработки изображений. Более подробное обсуждение трансформаций и линейной алгебры, лежащей в основе класса `AffineTransform`, можно найти в главе 4 книги «Java Foundation Classes in a Nutshell».

Пример 11.7. Transforms.java

```
package com.davidflanagan.examples.graphics;
import java.awt.*;
```

```
import java.awt.geom.*;

/** Показ трансформаций Java2D */
public class Transforms implements GraphicsExample {
 public String getName() { return "Transforms"; } // Из GraphicsExample
 public int getWidth() { return 750; }            // Из GraphicsExample
 public int getHeight() { return 250; }           // Из GraphicsExample

 Shape shape;                   // Изображаемая фигура
 AffineTransform[] transforms; // Способы ее трансформации
 String[] transformLabels;     // Подписи к трансформациям

 /**
 * Этот конструктор создает нужные нам объекты Shape и AffineTransform
 **/
 public Transforms() {
 GeneralPath path = new GeneralPath(); // Создаем отображаемую фигуру
 path.append(new Line2D.Float(0.0f, 0.0f, 0.0f, 100.0f), false);
 path.append(new Line2D.Float(-10.0f, 50.0f, 10.0f, 50.0f), false);
 path.append(new Polygon(new int[] { -5, 0, 5 },
    new int[] { 5, 0, 5 }, 3), false);
 this.shape = path; // Запоминаем эту фигуру

 // Задаем трансформации фигуры
 this.transforms = new AffineTransform[6];
 // 1) Тождественная (identity) трансформация
 transforms[0] = new AffineTransform();
 // 2) Трансформация масштабированием: 3/4 первоначального размера
 transforms[1] = AffineTransform.getScaleInstance(0.75, 0.75);
 // 3) Трансформация перекашиванием (shearing)
 transforms[2] = AffineTransform.getShearInstance(-0.4, 0.0);
 // 4) Вращение на 30 градусов по часовой стрелке относительно
       начальной точки (origin) фигуры
 transforms[3] = AffineTransform.getRotateInstance(Math.PI*2/12);
 // 5) Вращение на 180 градусов относительно центра фигуры
 transforms[4] = AffineTransform.getRotateInstance(Math.PI, 0.0, 50.0);
 // 6) Составная трансформация
 transforms[5] = AffineTransform.getScaleInstance(0.5, 1.5);
 transforms[5].shear(0.0, 0.4);
 transforms[5].rotate(Math.PI/2, 0.0, 50.0); // 90 degrees

 // Определяем названия трансформаций
 transformLabels = new String[] {
  "identity", "scale", "shear", "rotate", "rotate", "combo"
 };
 }

 /** Рисуем и подписываем преобразованную фигуру */
 public void draw(Graphics2D g, Component c) {
 // Определяем основные атрибуты рисования
 g.setColor(Color.black);                                    // черный
 g.setStroke(new BasicStroke(2.0f, BasicStroke.CAP_SQUARE, // 2 пиксела
     BasicStroke.JOIN_BEVEL));
 g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,       // сглаженный
    RenderingHints.VALUE_ANTIALIAS_ON);
```

```
 // Теперь рисуем фигуру, каждый раз применяя одну из определенных
 // выше трансформаций
 for(int i = 0; i < transforms.length; i++) {
  AffineTransform save = g.getTransform();    // сохраняем текущее состояние
  g.translate(i*125 + 50, 50);                // перемещаем отмеченную точку
  g.transform(transforms[i]);                 // применяем трансформацию
  g.draw(shape);                              // рисуем фигуру
  g.drawString(transformLabels[i], -25, 125); // подписываем метку
  g.drawRect(-40, -10, 80, 150);              // рисуем рамку
  g.setTransform(save);                      // восстанавливаем трансформацию
 }
 }
}
```

Задание стилей линий при помощи класса BasicStroke

В паре последних примеров для рисования линий толще одного пиксела, поддерживаемых классом `Graphics`, использовался класс `BasicStroke`. Однако толстые линии еще и сложнее тонких, поэтому `BasicStroke` позволяет задавать и другие атрибуты линий: *стиль концов* (*cap style*) линий указывает, как должны выглядеть концевые точки линий, а *стиль соединений* (*join style*) указывает, как должны выглядеть угловые точки или вершины фигур. Варианты стилей концевых точек и вершин показаны на рис. 11.6. Рисунок также демонстрирует применение штрих-пунктирной шаблонной линии – еще одной возможности класса `BasicStroke`. Рисунок создан при помощи кода, приведенного в примере 11.8, который показывает, как применять `BasicStroke` для рисования толстых линий с различными стилями концевых точек и соединений, а также для рисования шаблонных линий.

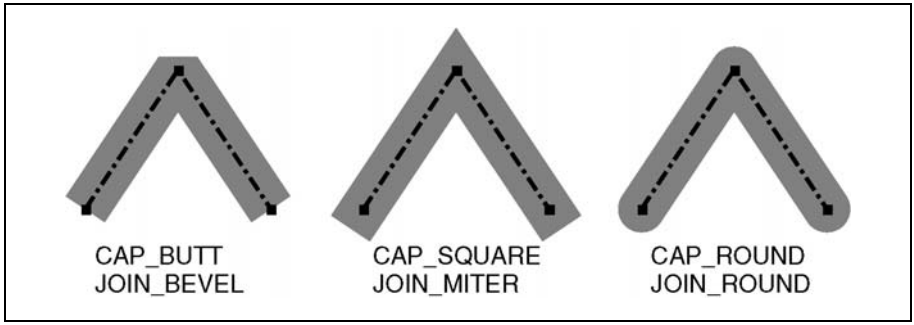


Рис. 11.6. *Стили линий от BasicStroke*

Пример 11.8. LineStyles.java

```
package com.davidflanagan.examples.graphics;
import java.awt.*;
```

```
import java.awt.geom.*;

/** Демонстрация стилей линий в Java2D */
public class LineStyles implements GraphicsExample {
 public String getName() { return "LineStyles"; } // Из GraphicsExample
 public int getWidth() { return 450; }            // Из GraphicsExample
 public int getHeight() { return 180; }           // Из GraphicsExample

 int[] xpoints = new int[] { 0, 50, 100 };       // Координаты X нашей фигуры
 int[] ypoints = new int[] { 75, 0, 75 };        // Координаты Y нашей фигуры

 // Здесь приводятся три разных стиля линий
 // Это толстые линии с различными стилями концевых точек и соединений
 Stroke[] linestyles = new Stroke[] {
 new BasicStroke(25.0f, BasicStroke.CAP_BUTT, BasicStroke.JOIN_BEVEL),
 new BasicStroke(25.0f, BasicStroke.CAP_SQUARE, BasicStroke.JOIN_MITER),
 new BasicStroke(25.0f, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND),
 };

 // Другой стиль линии: штрих-пунктирная линия толщиной в два пиксела
 Stroke thindashed = new BasicStroke(2.0f, // толщина линии
 /* стиль концов */   BasicStroke.CAP_BUTT,
 /* стиль соединения, предел среза */ BasicStroke.JOIN_BEVEL, 1.0f,
 /* шаблон штрих-пунктира */  new float[] {8.0f, 3.0f, 2.0f, 3.0f},
 /* фаза пунктира */  0.0f); /* есть 8, нет 3, есть 2, нет 3 */

 // Надписи, выводимые на диаграмме, и шрифт для их отображения.
 Font font = new Font("Helvetica", Font.BOLD, 12);
 String[] capNames = new String[] {"CAP_BUTT", "CAP_SQUARE","CAP_ROUND"};
 String[] joinNames = new String[] {"JOIN_BEVEL","JOIN_MITER","JOIN_ROUND"};

 /** Этот метод рисует фигуру из примера */
 public void draw(Graphics2D g, Component c) {
 // Применяем сглаживание чтобы, избежать "ступенек" в линиях
 g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
    RenderingHints.VALUE_ANTIALIAS_ON);

 // Определяем рисуемую фигуру
 GeneralPath shape = new GeneralPath();
 shape.moveTo(xpoints[0], ypoints[0]); // начинаем в точке 0
 shape.lineTo(xpoints[1], ypoints[1]); // проводим линию к точке 1
 shape.lineTo(xpoints[2], ypoints[2]); // и затем к точке 2

 // Перемещаем начальную точку, создавая поля
 g.translate(20,40);

 // Теперь в цикле рисуем фигуру, применяя три разных стиля
 for(int i = 0; i < linestyles.length; i++) {
  g.setColor(Color.gray);     // Рисуем серую линию
  g.setStroke(linestyles[i]); // Выбираем применяемый стиль
  g.draw(shape);              // Рисуем фигуру

  g.setColor(Color.black);    // Теперь черный цвет
  g.setStroke(thindashed);    // и тонкая пунктирная линия
  g.draw(shape);              // И снова рисуем линию.
```

```
    // Выделяем положение вершин фигур
    // Это подчеркивает демонстрируемые стили соединения и концов линий
    for(int j = 0; j < xpoints.length; j++)
    g.fillRect(xpoints[j]-2, ypoints[j]-2, 5, 5);

    g.drawString(capNames[i], 5, 105);  // Подписываем стиль концов
    g.drawString(joinNames[i], 5, 120); // Подписываем стиль соединений

    g.translate(150, 0); // Сдвигаемся направо перед очередным проходом цикла
   }
   }
  }
```

Рисование линий

Класс `BasicStroke` является реализацией интерфейса `Stroke`. Этот интерфейс в Java 2D отвечает за способ проведения (draw) или рисования (stroke) линий. Заливка цветом произвольных фигур – это фундаментальная графическая операция, определяемая в Java 2D. Интерфейс `Stroke` определяет API, при помощи которого операция рисования линии преобразуется в операцию заливки области, как это показано на рис. 11.7. Пример 11.9 показывает код, использованный для создания этого рисунка. (Поняв, как интерфейс `Stroke` преобразует фигуру в штриховую фигуру, пригодную для заливки, вы сможете определять собственные интересные реализации интерфейса `Stroke`, как мы это делаем в примере 11.15 этой главы.)

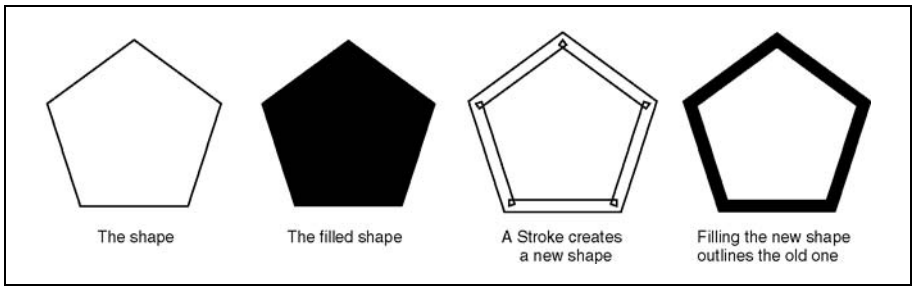


***Рис. 11.7.** Рисование линий в Java 2D*

Пример 11.9. Stroking.java

```
package com.davidflanagan.examples.graphics;
import java.awt.*;
import java.awt.geom.*;

/** Демонстрация работы объекта Stroke */
public class Stroking implements GraphicsExample {
 static final int WIDTH = 725, HEIGHT = 250;  // Размер нашего графического
                                              // примера
 public String getName() {return "Stroking";} // Из GraphicsExample
```

```
public int getWidth() { return WIDTH; }      // Из GraphicsExample
public int getHeight() { return HEIGHT; }    // Из GraphicsExample

/** Рисуем пример */
public void draw(Graphics2D g, Component c) {
// Создаем фигуру, с которой будем работать. См. вспомогательный метод ниже.
Shape pentagon = createRegularPolygon(5, 75);

// Устанавливаем основные атрибуты рисования
g.setColor(Color.black);                          // Рисуем черным
g.setStroke(new BasicStroke(1.0f));               // Используем тонкие линии
g.setFont(new Font("SansSerif", Font.PLAIN, 12)); // Основной мелкий шрифт

g.translate(100, 100);                            // Перемещаемся на место
g.draw(pentagon);                                 // Вычерчиваем фигуру
g.drawString("The shape", -30, 90);               // Выводим надпись

g.translate(175, 0);                              // Перемещаемся
g.fill(pentagon);                                 // Заливаем фигуру
g.drawString("The filled shape", -50, 90);        // Другая надпись

// Теперь используем объект Stroke для создания из нашей фигуры
// "штриховой фигуры" (stroked shape)[1]
BasicStroke wideline = new BasicStroke(10.0f);
Shape outline = wideline.createStrokedShape(pentagon);

g.translate(175, 0);                      // Перемещаемся
g.draw(outline);                          // Выводим штриховую фигуру
g.drawString("A Stroke creates",-50,90); // Выводим надпись
g.drawString("a new shape", -35, 105);

g.translate(175,0);                           // Перемещаемся
g.fill(outline);                              // Заливаем контур фигуры
g.drawString("Filling the new shape",-65,90); // Выводим надпись
g.drawString("outlines the old one",-65,105);
}

// Вспомогательный метод для определения правильного многоугольника.
// Возвращает фигуру, представляющую правильный многоугольник с заданными
// радиусом и числом сторон и с началом в центре.
public Shape createRegularPolygon(int numsides, int radius) {
Polygon p = new Polygon();
double angle = 2 * Math.PI / numsides; // Угол между вершинами
for(int i = 0; i < numsides; i++)      // Вычисляем положение каждой вершины
 p.addPoint((int)(radius * Math.sin(angle*i)),
  (int)(radius * -Math.cos(angle*i)));
return p;
}
}
```

[1] Здесь выражение «штриховая фигура» означает «фигура, контур которой обведен штрихом»; «штрихом» (stroke) называется ширина (форма) следа, оставляемого пером (или кистью) при рисовании. – *Примеч. ред.*

Заливка фигур при помощи классов Paint

Мы только что видели, что рисование линий в Java 2D определяется в терминах более фундаментальной операции заливки области. В предыдущих примерах мы использовали метод `fill()` класса `Graphics2D` для заливки внутренней области объекта `Shape` произвольным сплошным цветом, который до этого был передан методу `setColor()`. Java 2D API, однако, обобщает понятие цвета и допускает заливку области при помощи любой реализации интерфейса `Paint`. Реализация `Paint` отвечает за задание цветов, используемых при операции заливки области (или рисования линий). Начиная с Java 1.2 класс `Color` реализует метод интерфейса `Paint`, который позволяет заливать область сплошным цветом. Java 2D определяет также две другие реализации `Paint`: `GradientPaint` закрашивает фигуру цветовым градиентом, а `TexturePaint` заливает фигуру копиями изображения-«плитки» (tile). Эти классы можно использовать для получения эффектов заливки, показанных на рис. 11.8.

***Рис. 11.8.** Заливка фигур при помощи объектов Paint*

Пример 11.10 представляет код, использованный для создания рис. 11.8. Помимо классов `GradientPaint` и `TexturePaint`, этот пример демонстрирует многие другие возможности Java 2D: прозрачные цвета, представление букв как объектов `Shape` для создания «художественного текста», сочетание трансформаций класса `AffineTransform` и прозрачных цветов для создания «теней», а также применение класса `BufferedImage` для внеэкранного рисования. Пример 11.10 иллюстрирует также применение класса `GenericPaint`, собственной (custom) реализации интерфейса `Paint`, которую мы увидим в примере 11.16.

Пример 11.10. Paints.java

```
package com.davidflanagan.examples.graphics;
import java.awt.*;
```

```
import java.awt.geom.*;
import java.awt.font.*;
import java.awt.image.*;

/** Демонстрация использования Java2D для создания художественного текста */
public class Paints implements GraphicsExample {
 static final int WIDTH = 800, HEIGHT = 375;  // Размеры нашего
                                              // графического примера

 public String getName() { return "Paints"; } // Из GraphicsExample
 public int getWidth() { return WIDTH; }      // Из GraphicsExample
 public int getHeight() { return HEIGHT; }    // Из GraphicsExample

 /** Рисуем пример */
 public void draw(Graphics2D g, Component c) {
 // Общий фон рисуем при помощи объекта GradientPaint.
 // Фоновый цвет изменяется по диагонали от темно-красного до бледно-голубого
 g.setPaint(new GradientPaint(0, 0, new Color(150, 0, 0),
     WIDTH, HEIGHT, new Color(200, 200, 255)));
 g.fillRect(0, 0, WIDTH, HEIGHT);  // заливаем фон

 // Используем другой градиент GradientPaint для рисования рамки.
 // Это переход от непрозрачного темно-зеленого до прозрачного зеленого.
 // Замечание: четвертый аргумент конструктора Color() задает
 // непрозрачность (opacity) цвета
 g.setPaint(new GradientPaint(0, 0, new Color(0, 150, 0),
     20, 20, new Color(0, 150, 0, 0), true));
 g.setStroke(new BasicStroke(15));        // применяем толстые линии
 g.drawRect(25, 25, WIDTH-50, HEIGHT-50); // рисуем рамку

 // Символы шрифтов могут использоваться как объекты Shape, что позволяет
 // применять приемы Java2D к буквам, как к любым другим фигурам.
 // Здесь получаем некоторые буквы для рисования.
 Font font = new Font("Serif", Font.BOLD, 10); // Основной шрифт
 Font bigfont =                                // Его увеличенный вариант
  font.deriveFont(AffineTransform.getScaleInstance(30.0, 30.0));
 GlyphVector gv = bigfont.createGlyphVector(g.getFontRenderContext(),
       "JAV");
 Shape jshape = gv.getGlyphOutline(0); // Фигура буквы J
 Shape ashape = gv.getGlyphOutline(1); // Фигура буквы A
 Shape vshape = gv.getGlyphOutline(2); // Фигура буквы V

 // Мы собираемся обвести буквы линией толщиной в пять пикселов
 g.setStroke(new BasicStroke(5.0f));

 // Мы собираемся сымитировать тени от букв при помощи
 // следующих объектов Paint и AffineTransform
 Paint shadowPaint = new Color(0, 0, 0, 100);  // Прозрачный черный
 AffineTransform shadowTransform =
  AffineTransform.getShearInstance(-1.0, 0.0); // Перекашиваем вправо
 shadowTransform.scale(1.0, 0.5);              // Уменьшаем высоту вдвое

 // Передвигаемся к опорной линии первой буквы
 g.translate(65, 270);
```

```
// Рисуем тень фигуры J
g.setPaint(shadowPaint);
g.translate(15,20); // Учитываем подстрочный элемент буквы J,
// преобразуем J по форме ее тени и заливаем ее
g.fill(shadowTransform.createTransformedShape(jshape));
g.translate(-15,-20); // Отменяем предыдущий сдвиг

// Теперь заливаем фигуру J сплошным (и непрозрачным) цветом
g.setPaint(Color.blue);  // Устанавливаем заливку сплошным непрозрачным цветом
g.fill(jshape);          // Заливаем фигуру буквы
g.setPaint(Color.black); // Переключаемся на сплошной черный
g.draw(jshape);          // и обводим контур фигуры J

// Теперь рисуем тень буквы A
g.translate(75, 0);      // Сдвигаемся вправо
g.setPaint(shadowPaint); // Устанавливаем цвет тени
g.fill(shadowTransform.createTransformedShape(ashape)); // Рисуем тень

// Рисуем A с применением сплошного прозрачного цвета
g.setPaint(new Color(0, 255, 0, 125)); // Прозрачный зеленый
g.fill(ashape);                        // Заливаем фигуру буквы
g.setPaint(Color.black);               // Переключаемся на сплошной черный
g.draw(ashape);                        // Рисуем контур буквы

// Передвигаемся вправо и рисуем тень буквы V
g.translate(175, 0);
g.setPaint(shadowPaint);
g.fill(shadowTransform.createTransformedShape(vshape));

// Мы собираемся залить следующую букву, используя TexturePaint, который
// периодически повторяет изображение. Первым делом нужно получить
// изображение "плитки". Можно было бы загрузить рисунок из файла, но здесь
// мы сами создадим его, нарисовав вне экрана. Обратите внимание, что мы
// используем GradientPaint для заливки внеэкранного изображения,
// так что в образце заливки сочетаются возможности обоих классов Paint.
BufferedImage tile =                   // Создаем изображение "плитки"
 new BufferedImage(50, 50, BufferedImage.TYPE_INT_RGB);
Graphics2D tg = tile.createGraphics(); // Получаем его объект Graphics
                                       // для рисования
tg.setColor(Color.pink);
tg.fillRect(0, 0, 50, 50); // Заливаем фон "плитки" розовым
tg.setPaint(new GradientPaint(40, 0, Color.green, // Диагональный градиент
    0, 40, Color.gray));   // от зеленого к серому
tg.fillOval(5, 5, 40, 40); // Рисуем кружок с этим градиентом

// Применяем созданную "плитку" для создания TexturePaint и заполнения буквы V
g.setPaint(new TexturePaint(tile, new Rectangle(0, 0, 50, 50)));
g.fill(vshape);           // Заливаем фигуру буквы
g.setPaint(Color.black);  // Переключаемся на сплошной черный
g.draw(vshape);           // Рисуем контур буквы

// Перемещаемся вправо и рисуем тень последней буквы A
g.translate(160, 0);
g.setPaint(shadowPaint);
```

```
        g.fill(shadowTransform.createTransformedShape(ashape));

        // Для заливки последней буквы используем собственный класс Paint
        // со сложной математически определенной моделью. Класс GenericPaint
        // определен ниже в этой главе.
        g.setPaint(new GenericPaint() {
         public int computeRed(double x, double y) { return 128; }
         public int computeGreen(double x, double y) {
          return (int)((Math.sin(x/7) + Math.cos(y/5) + 2)/4 *255);
         }
         public int computeBlue(double x, double y) {
          return ((int)(x*y))%256;
         }
         public int computeAlpha(double x, double y) {
          return ((int)x%25*8+50) + ((int)y%25*8+50);
         }
         });
        g.fill(ashape);           // Заливаем букву A
        g.setPaint(Color.black); // Возвращаемся к сплошному черному
        g.draw(ashape);           // Рисуем контур буквы A
        }
    }
```

Сглаживание

Как мы уже видели, при рисовании текста и графики можно потребовать, чтобы Java 2D выполнял сглаживание. Сглаживанию подвергаются линии и границы фигур (таких как символы шрифтов), в результате чего уменьшаются видимая ступенчатость. Сглаживание при рисовании необходимо из-за того, что контур фигуры на экране монитора не может быть проведен совершенно ровно; идеальная математическая фигура не может быть абсолютно точно перенесена на дискретную пиксельную решетку. При рисовании фигуры пикселы внутри нее заливаются, а пикселы снаружи нее – нет. Контур фигуры, однако, редко точно попадает на границу между пикселами, поэтому края фигуры приблизительны. В результате получаются ступенчатые линии, приблизительно похожие на абстрактную фигуру, которую мы хотим представить.

Сглаживание представляет собой просто технику улучшения вида этого приближения при помощи прозрачных цветов. Если, например, пиксел на краю фигуры покрыт фигурой наполовину, пиксел заполняется полупрозрачным цветом. Если покрыта только одна пятая часть пиксела, пиксел делается прозрачным на одну пятую. Эта техника вполне подходит для уменьшения видимых ступенек. Рисунок 11.9 иллюстрирует процесс сглаживания: он показывает сглаженный рисунок в искусственно увеличенном виде, чтобы стали видны прозрачные цвета, использованные на краях фигур и элементов шрифта. Рисунок создан с помощью простого кода из примера 11.11.

Рис. 11.9. *Сглаживание с увеличением*

Пример 11.11. AntiAlias.java

```
package com.davidflanagan.examples.graphics;
import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;

/** Демонстрация сглаживания */
public class AntiAlias implements GraphicsExample {
 static final int WIDTH = 650, HEIGHT = 350;      // Размеры нашего
                                                  // графического примера
 public String getName() {return "AntiAliasing";} // Из GraphicsExample
 public int getWidth() { return WIDTH; }          // Из GraphicsExample
 public int getHeight() { return HEIGHT; }        // Из GraphicsExample

 /** Рисуем пример */
 public void draw(Graphics2D g, Component c) {
 BufferedImage image =                    // Создаем внеэкранное изображение
  new BufferedImage(65, 35, BufferedImage.TYPE_INT_RGB);
 Graphics2D ig = image.createGraphics(); // Получаем его объект Graphics
                                         // для рисования

 // Устанавливаем плавное изменение фонового цвета. Изменение
 // цвета фона помогает продемонстрировать эффект сглаживания
 ig.setPaint(new GradientPaint(0,0,Color.black,65,35,Color.white));
 ig.fillRect(0, 0, 65, 35);

 // Устанавливаем атрибуты рисования для переднего плана.
 // Самое важное – включаем сглаживание.
 ig.setStroke(new BasicStroke(2.0f));         // Линии толщиной в два пиксела
 ig.setFont(new Font("Serif", Font.BOLD, 18)); // Шрифт размером 18 пунктов
 ig.setRenderingHint(RenderingHints.KEY_ANTIALIASING, // Сгладить!
```

```
        RenderingHints.VALUE_ANTIALIAS_ON);

    // Теперь рисуем текст чистого синего цвета и овал чистого красного цвета
    ig.setColor(Color.blue);
    ig.drawString("Java", 9, 22);
    ig.setColor(Color.red);
    ig.drawOval(1, 1, 62, 32);

    // Наконец, масштабируем изображение с коэффициентом 10 и отображаем его
    // в окне. Так будет видно сглаженные пикселы.
    g.drawImage(image, AffineTransform.getScaleInstance(10, 10), c);

    // Для сравнения еще раз рисуем изображение, уже в натуральную величину
    g.drawImage(image, 0, 0, c);
    }
}
```

Комбинирование цветов при помощи AlphaComposite

Как я только что объяснил, сглаживание осуществляется при помощи рисования краев фигур прозрачными цветами. Но что же конкретно понимается под рисованием прозрачными цветами? Посмотрите еще раз на рис. 11.9 или, еще лучше, запустите этот пример при помощи программы `GraphicsExampleFrame`, чтобы увидеть все в цвете. При рисовании прозрачным цветом нижележащий цвет «просвечивает» сквозь него. На рис. 11.9 фоновые серые цвета просвечивают сквозь чистые прозрачные красный и синий цвета, в результате чего получаются синеватый и красноватый оттенки серого. На аппаратном уровне, разумеется, нет никаких прозрачных цветов; рисование прозрачным цветом имитируется сочетанием цвета рисунка и цвета подложки.

Такого рода сочетание цветов называется их *композицией* и осуществляется интерфейсом `Composite`. Можно передать объект `Composite` методу `setComposite()` объекта `Graphics2D`, чтобы сообщить ему, как следует комбинировать цвет рисунка (цвет источника) с цветами, уже находящимися на поверхности рисунка (цветами приемника). Java 2D определяет одну реализацию интерфейса `Composite`, `AlphaComposite`, которая комбинирует цвета, основываясь на их уровнях прозрачности.

Принимаемый объектом `Graphics2D` по умолчанию объект `AlphaComposite` подходит для большинства рисунков, поэтому создавать собственные объекты `AlphaComposite` приходится редко. Тем не менее при помощи класса `AlphaComposite` можно получить интересные эффекты. Рисунок 11.10 демонстрирует такие эффекты (и не относящийся к ним эффект обрезки), полученные при помощи кода из примера 11.12.

В этом примере большая часть рисования производится на внеэкранном изображении, после чего содержимое изображения копируется на экран. Причина в том, что многие эффекты композиции могут быть полу-

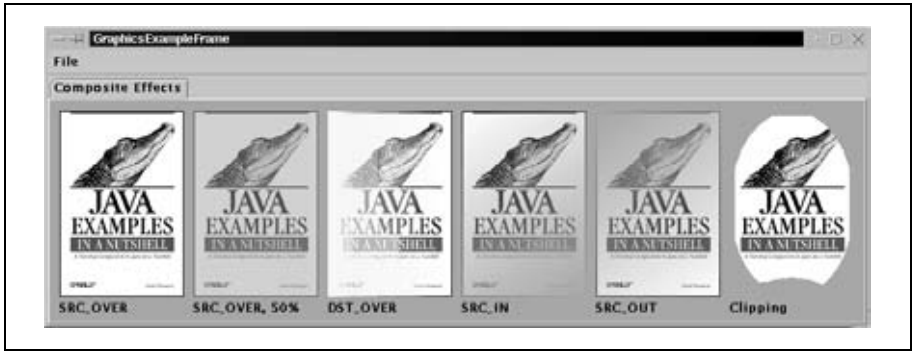


***Рис. 11.10.** Эффекты, созданные при помощи AlphaComposite*

чены только при работе с такой поверхностью рисования, которая обладает альфа-каналом («alpha channel») и поддерживает прозрачные цвета (например, внеэкранное изображение). Убедитесь, что разобрались, как это внеэкранное изображение создано при помощи `BufferedImage`.

Пример 11.12 иллюстрирует также тип эффектов, получаемых при задании области обрезки. Java 2D позволяет использовать любой объект `Shape` для задания области обрезки; графика отображается, только если она попадает в эту область. В примере используется класс `java.awt.geom.Area` для определения сложной фигуры путем комбинации двух эллипсов и прямоугольника, затем эта фигура используется как область обрезки.

Пример 11.12. CompositeEffects.java

```
package com.davidflanagan.examples.graphics;
import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;

public class CompositeEffects implements GraphicsExample {
 Image cover; // Изображение, которое мы будем выводить, и его размеры
 static final int COVERWIDTH = 127, COVERHEIGHT = 190;

 /** Этот конструктор загружает изображение с обложки */
 public CompositeEffects() {
 java.net.URL imageurl = this.getClass().getResource("cover.gif");
 cover = new javax.swing.ImageIcon(imageurl).getImage();
 }

 // Это основные методы GraphicsExample
 public String getName() {return "Composite Effects";}
 public int getWidth() { return 6*COVERWIDTH + 70; }
 public int getHeight() { return COVERHEIGHT + 35; }

 /** Рисуем пример */
 public void draw(Graphics2D g, Component c) {
 // заливка фона
```

```
g.setPaint(new Color(175, 175, 175));
g.fillRect(0, 0, getWidth(), getHeight());

// Устанавливаем атрибуты текста
g.setColor(Color.black);
g.setFont(new Font("SansSerif", Font.BOLD, 12));

// Рисуем исходное изображение
g.translate(10, 10);
g.drawImage(cover, 0, 0, c);
g.drawString("SRC_OVER", 0, COVERHEIGHT+15);

// Рисуем обложку снова, используя AlphaComposite, чтобы сделать
// непрозрачные цвета изображения прозрачными на 50%
g.translate(COVERWIDTH+10, 0);
g.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER,
      0.5f));
g.drawImage(cover, 0, 0, c);

// Восстанавливаем заранее определенные принимаемые по умолчанию свойства
// Composite для экрана, так что непрозрачные цвета остаются непрозрачными.
g.setComposite(AlphaComposite.SrcOver);
// Надписываем эффект
g.drawString("SRC_OVER, 50%", 0, COVERHEIGHT+15);

// Теперь создаем внеэкранное изображение, с которым будем работать. Чтобы
// получались определенные эффекты композиции, поверхность рисунка должна
// поддерживать прозрачность. Экранные поверхности рисунка этого не делают,
// поэтому придется осуществлять композицию на внеэкранном изображении,
// специально созданном, чтобы имелся альфа-канал, а затем копировать
// окончательный результат на экран.
BufferedImage offscreen =
 new BufferedImage(COVERWIDTH, COVERHEIGHT,
   BufferedImage.TYPE_INT_ARGB);

// Во-первых, заполняем фон изображения цветовым градиентом, который
// меняется слева направо от непрозрачного до прозрачного желтого
Graphics2D osg = offscreen.createGraphics();
osg.setPaint(new GradientPaint(0, 0, Color.yellow,
    COVERWIDTH, 0,
    new Color(255, 255, 0, 0)));
osg.fillRect(0,0, COVERWIDTH, COVERHEIGHT);

// Теперь копируем поверх него изображение обложки, но применяем правило
// DstOver, по которому рисование производится «под» существующими
// пикселами, сквозь которые допускается просвечивание рисунка
// в меру их прозрачности.
osg.setComposite(AlphaComposite.DstOver);
osg.drawImage(cover, 0, 0, c);

// И выводим это композитное изображение на экран. Обратите внимание на то,
// что изображение непрозрачное, и фон экрана сквозь него не просвечивает.
g.translate(COVERWIDTH+10, 0);
g.drawImage(offscreen, 0, 0, c);
```

```
g.drawString("DST_OVER", 0, COVERHEIGHT+15);

// Опять начинаем и создаем новый эффект при помощи внеэкранного
// изображения. Во-первых, заполняем изображение новым цветовым градиентом.
// Нас не заботят сами цвета; нам нужно только, чтобы изменялась прозрачность
// фона. Мы используем градиент от непрозрачного черного к прозрачному
// черному. Обратите внимание, что поскольку мы уже использовали это
// изображение, мы назначаем правило композиции Src и можем заполнять
// изображение, игнорируя то, что там уже находится.
osg.setComposite(AlphaComposite.Src);
osg.setPaint(new GradientPaint(0, 0, Color.black,
    COVERWIDTH, COVERHEIGHT,
    new Color(0, 0, 0, 0)));
osg.fillRect(0,0, COVERWIDTH, COVERHEIGHT);

// Теперь устанавливаем тип композиции в SrcIn, так что цвета определяются
// источником, а прозрачность - свойствами приемника
osg.setComposite(AlphaComposite.SrcIn);

// Рисуем загруженное изображение на внеэкранном изображении с композицией.
osg.drawImage(cover, 0, 0, c);

// И теперь копируем внеэкранное изображение на экран. Обратите внимание
// на то, что изображение прозрачно, и сквозь него проступает фон.
g.translate(COVERWIDTH+10, 0);
g.drawImage(offscreen, 0, 0, c);
g.drawString("SRC_IN", 0, COVERHEIGHT+15);

// Если проделать все то же с применением SrcOut, результирующее изображение
// будет иметь инвертированные значения прозрачности в сравнении
// со свойствами приемника
osg.setComposite(AlphaComposite.Src);
osg.setPaint(new GradientPaint(0, 0, Color.black,
    COVERWIDTH, COVERHEIGHT,
    new Color(0, 0, 0, 0)));
osg.fillRect(0,0, COVERWIDTH, COVERHEIGHT);
osg.setComposite(AlphaComposite.SrcOut);
osg.drawImage(cover, 0, 0, c);
g.translate(COVERWIDTH+10, 0);
g.drawImage(offscreen, 0, 0, c);
g.drawString("SRC_OUT", 0, COVERHEIGHT+15);

// Здесь достигается яркий эффект; он не имеет отношения к композиции,
// а применяет произвольную фигуру для обрезки изображения.
// Здесь используется класс Area для комбинирования
// фигур с целью создания более сложных фигур.
g.translate(COVERWIDTH+10, 0);
Shape savedClip = g.getClip(); // сохраняем текущую область обрезки
// Создаем фигуру, которая будет использоваться как новая область обрезки.
// Начинаем с эллипса
Area clip = new Area(new Ellipse2D.Float(0,0,COVERWIDTH,COVERHEIGHT));
// Применяем операцию пересечения с прямоугольником, обрезая эллипс.
clip.intersect(new Area(new Rectangle(5,5,
     COVERWIDTH-10,COVERHEIGHT-10)));
```

```
 // Теперь выполняем вычитание эллипса из нижней части обрезанного эллипса.
 clip.subtract(new Area(new Ellipse2D.Float(COVERWIDTH/2-40,
       COVERHEIGHT-20, 80, 40)));
 // Используем полученную фигуру в качестве новой области обрезки
 g.clip(clip);
 // Теперь рисуем изображение с этой областью обрезки.
 g.drawImage(cover, 0, 0, c);
 // Восстанавливаем прежнюю область обрезки, чтобы иметь
 // возможность подписать эффект.
 g.setClip(savedClip);
 g.drawString("Clipping", 0, COVERHEIGHT+15);
 }
}
```

Обработка изображений

И Java 1.0, и Java 1.1 включали сложный API для фильтрования изображений «на лету», то есть в процессе их загрузки по сети. Хотя этот API остается доступным и в более поздних версиях Java, обычно он не используется и не рассматривается в этой книге. Java 2D определяет более простой API, основанный на интерфейсе `BufferedImageOp` **из пакета** `java.awt.image`**. Этот пакет также содержит несколько универсальных реализаций этого интерфейса, с помощью которых можно получать эффекты обработки изображений, проиллюстрированные на рис. 11.11. В примере 11.13 показан код, использованный для создания этого рисунка. Код легко понять: для обработки изображения** `BufferedImage` **просто передается методу** `filter()` **интерфейса** `BufferedImageOp`**.**

Пример 11.13. ImageOps.java

```
package com.davidflanagan.examples.graphics;
import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;
import java.awt.color.*;

/** Демонстрация различных фильтров обработки изображений */
public class ImageOps implements GraphicsExample {
 static final int WIDTH = 600, HEIGHT = 675;          // Размеры нашего
                                                      // графического примера
 public String getName() {return "Image Processing";} // Из GraphicsExample
 public int getWidth() { return WIDTH; }              // Из GraphicsExample
 public int getHeight() { return HEIGHT; }            // Из GraphicsExample

 Image image;

 /** Этот конструктор загружает изображение, которым мы будем манипулировать */
 public ImageOps() {
 java.net.URL imageurl = this.getClass().getResource("cover.gif");
 image = new javax.swing.ImageIcon(imageurl).getImage();
 }
```

Рис. 11.11. *Обработка изображений при помощи BufferedImageOp*

```
// Эти массивы байтов используются ниже фильтрами изображений LookupImageOp
static byte[] brightenTable = new byte[256];
static byte[] thresholdTable = new byte[256];
static { // Инициализируем массивы
for(int i = 0; i < 256; i++) {
 brightenTable[i] = (byte)(Math.sqrt(i/255.0)*255);
 thresholdTable[i] = (byte)((i < 225)?0:i);
}
}

// Этот объект AffineTransform используется ниже одним
// из фильтров изображений
static AffineTransform mirrorTransform;
static { // Создаем и инициализируем AffineTransform
```

```
mirrorTransform = AffineTransform.getTranslateInstance(127, 0);
mirrorTransform.scale(-1.0, 1.0); // отражаем по горизонтали
}

// Это надписи, которые будут отображаться ниже изображений,
// обработанных фильтрами
static String[] filterNames = new String[] {
"Original", "Gray Scale", "Negative", "Brighten (linear)",
"Brighten (sqrt)", "Threshold", "Blur", "Sharpen",
"Edge Detect", "Mirror", "Rotate (center)", "Rotate (lower left)"
};

// Следующие объекты фильтров изображений BufferedImageOp выполняют
// различные операции обработки изображений.
static BufferedImageOp[] filters = new BufferedImageOp[] {
// 1) Фильтра нет. Мы отобразим оригинальное изображение
null,
// 2) Преобразуем к пространству цветов Grayscale (оттенки серого)
new ColorConvertOp(ColorSpace.getInstance(ColorSpace.CS_GRAY), null),
// 3) Получаем негатив изображения. Умножаем каждое значение цвета
      на -1.0 и прибавляем 255
new RescaleOp(-1.0f, 255f, null),
// 4) Делаем ярче, используя линейную формулу, которая увеличивает
      все значения цветов
new RescaleOp(1.25f, 0, null),
// 5) Делаем ярче, используя справочную таблицу, определенную выше
new LookupOp(new ByteLookupTable(0, brightenTable), null),
// 6) "Обрезаем" цвета, используя определенную выше таблицу
      пороговых значений
new LookupOp(new ByteLookupTable(0, thresholdTable), null),
// 7) Делаем изображение размытым, усредняя цвета при помощи матрицы
new ConvolveOp(new Kernel(3, 3, new float[] {
 .1111f,.1111f,.1111f,
 .1111f,.1111f,.1111f,
 .1111f,.1111f,.1111f,})),
// 8) При помощи другой матрицы делаем изображение контрастней
new ConvolveOp(new Kernel(3, 3, new float[] {
 0.0f, -0.75f, 0.0f,
 -0.75f, 4.0f, -0.75f,
 0.0f, -0.75f, 0.0f})),
// 9) Выделяем края при помощи еще одной матрицы
new ConvolveOp(new Kernel(3, 3, new float[] {
 0.0f, -0.75f, 0.0f,
 -0.75f, 3.0f, -0.75f,
 0.0f, -0.75f, 0.0f})),
// 10) Вычисляем зеркальное изображение, используя определенное
       выше преобразование
new AffineTransformOp(mirrorTransform,AffineTransformOp.TYPE_BILINEAR),
// 11) Вращаем изображение на 180 градусов относительно его центра
new AffineTransformOp(AffineTransform.getRotateInstance(Math.PI,64,95),
   AffineTransformOp.TYPE_NEAREST_NEIGHBOR),
// 12) Вращаем изображение на 15 градусов относительно его левого нижнего угла
```

```
new AffineTransformOp(AffineTransform.getRotateInstance(Math.PI/12,
       0, 190),
   AffineTransformOp.TYPE_NEAREST_NEIGHBOR),
};

/** Рисуем пример */
public void draw(Graphics2D g, Component c) {
// Создаем BufferedImage достаточно большой, чтобы вместить загруженное
// в конструкторе изображение. Затем копируем это изображение в новый объект
// BufferedImage, чтобы иметь возможность его обработать.
BufferedImage bimage = new BufferedImage(image.getWidth(c),
      image.getHeight(c),
      BufferedImage.TYPE_INT_RGB);
Graphics2D ig = bimage.createGraphics();
ig.drawImage(image, 0, 0, c); // копируем изображение

// Устанавливаем некоторые атрибуты графики
g.setFont(new Font("SansSerif", Font.BOLD, 12)); // жирный шрифт размером
                                                 // 12 пунктов
g.setColor(Color.green);                         // Рисуем зеленым
g.translate(10, 10);                             // Делаем отступ от края

// Цикл по фильтрам
for(int i = 0; i < filters.length; i++) {
 // Если фильтр пустой, рисуем первоначальное изображение, в противном
 // случае выводим изображение, обработанное фильтром.
 if (filters[i] == null) g.drawImage(bimage, 0, 0, c);
 else g.drawImage(filters[i].filter(bimage, null), 0, 0, c);
 g.drawString(filterNames[i], 0, 205);     // Подписываем изображение
 g.translate(137, 0);                      // Смещаемся
 if (i % 4 == 3) g.translate(-137*4, 215); // Смещаемся вниз после каждого
                                           // четвертого изображения
}
}
}
```

Пользовательские фигуры

Пример 11.4 показывал, как может применяться Java 2D для рисования и заполнения фигур разного вида. Одной из приведенных фигур была спираль, которая рисовалась при помощи класса `Spiral`, пользовательской реализации интерфейса `Shape`, показанной в примере 11.14.

Интерфейс `Shape` определяет три важных метода (часть из них имеют несколько перегруженных (overload) версий), которые должны реализовать все фигуры. Методы `contains()` определяют, содержит ли фигура точку или прямоугольник; объект `Shape` должен уметь отличать свою внутреннюю область от внешней. Поскольку наша спираль является незамкнутой кривой, у нее нет внутренней области, и эти методы всегда возвращают `false`. Методы `intersects()` определяют, пересекается ли какая-то часть фигуры с заданным прямоугольником. Посколь-

ку для спирали это трудно вычислить точно, для этих методов программа аппроксимирует спираль окружностью.

Методы `getPathIterator()` составляют ядро любой реализации `Shape`. Каждый метод возвращает объект `PathIterator`, описывающий контур фигуры в терминах сегментов линий и кривых. Java 2D использует объекты `PathIterator` при рисовании и заполнении фигур. Ключевыми методами в реализации `SpiralIterator` являются метод `currentSegment()`, возвращающий один прямой отрезок спирали, и метод `next()`, переводящий итератор к следующему отрезку. Чтобы обеспечить достаточно хорошую аппроксимацию спирали отрезками прямых, метод `next()` использует довольно сложный математический аппарат.

Пример 11.14. Spiral.java

```
package com.davidflanagan.examples.graphics;
import java.awt.*;
import java.awt.geom.*;

/** Эта реализация Shape представляет спиральную кривую */
public class Spiral implements Shape {
 double centerX, centerY;         // Центр спирали
 double startRadius, startAngle; // Начальная точка спирали
 double endRadius, endAngle;      // Конечная точка спирали
 double outerRadius;              // Внешний радиус
 int angleDirection;     // 1, если угол увеличивается, -1 в противном случае

 /**
 * Конструктор. Он принимает аргументы, задающие центр фигуры, а также
 * начальную и конечную точки. Начальная и конечная точки задаются в терминах
 * угла и радиуса. Спиральная кривая формируется путем плавного
 * изменения угла и радиуса между начальной и конечной точками.
 **/
 public Spiral(double centerX, double centerY,
   double startRadius, double startAngle,
   double endRadius, double endAngle)
 {
 // Сохраняем описывающие спираль параметры
 this.centerX = centerX;  this.centerY = centerY;
 this.startRadius = startRadius; this.startAngle = startAngle;
 this.endRadius = endRadius; this.endAngle = endAngle;

 // вычисляем максимальный радиус и направление спирали
 this.outerRadius = Math.max(startRadius, endRadius);
 if (startAngle < endAngle) angleDirection = 1;
 else angleDirection = -1;

 if ((startRadius < 0) || (endRadius < 0))
  throw new IllegalArgumentException("Spiral radii must be >= 0");
 }

 /**
 * Ограничивающий прямоугольник спирали будет таким же, как у окружности
```

```
* с тем же центром и максимальным радиусом спирали
**/
public Rectangle getBounds() {
return new Rectangle((int)(centerX-outerRadius),
   (int)(centerY-outerRadius),
   (int)(outerRadius*2), (int)(outerRadius*2));
}

/** То же, что getBounds(), но координаты с плавающей точкой */
public Rectangle2D getBounds2D() {
return new Rectangle2D.Double(centerX-outerRadius, centerY-outerRadius,
    outerRadius*2, outerRadius*2);
}

/**
* Спираль – это открытая, незамкнутая кривая; у нее нет внешней
* и внутренней областей, поэтому методы contains() всегда возвращают false.
**/
public boolean contains(double x, double y) { return false; }
public boolean contains(Point2D p) { return false; }
public boolean contains(Rectangle2D r) { return false; }
public boolean contains(double x, double y, double w, double h) {
return false;
}

/**
* Этот метод допустимо использовать, если вычисление точного результата
* оказывается слишком долгим. Так что мы проверяем здесь, пересекается ли
* прямоугольник с окружностью внешнего радиуса. Это хорошо подходит
* для "туго закрученной" спирали, но хуже для "свободной" спирали.
**/
public boolean intersects(double x, double y, double w, double h) {
Shape approx = new Ellipse2D.Double(centerX-outerRadius,
     centerY-outerRadius,
     outerRadius*2, outerRadius*2);
return approx.intersects(x, y, w, h);
}

/** Эта версия метода intersects() просто вызывает предыдущий метод */
public boolean intersects(Rectangle2D r) {
return intersects(r.getX(), r.getY(), r.getWidth(), r.getHeight());
}

/**
* Этот метод является ядром любой реализации Shape. Он возвращает
* PathIterator, описывающий фигуру в терминах составляющих ее отрезков
* прямых и кривых. Наша реализация итератора аппроксимирует спираль,
* используя только отрезки прямых. Его конструктору мы передаем аргумент
* flatness, который сообщает ему, насколько хорошим должно быть приближение.
* (Чем меньше значение flatness, тем лучше приближение).
*/
public PathIterator getPathIterator(AffineTransform at) {
return new SpiralIterator(at, outerRadius/500.0);
```

```
}

/**
* Возвращаем PathIterator, описывающий фигуру в терминах отрезков прямых
* с качеством приближения, задаваемым аргументом flatness.
**/
public PathIterator getPathIterator(AffineTransform at, double flatness) {
return new SpiralIterator(at, flatness);
}

/**
* Этот внутренний класс является итератором PathIterator для фигуры Spiral.
* Для простоты он описывает спираль не в терминах сегментов кривых Безье,
* а просто аппроксимирует ее отрезками прямых. Свойство flatness указывает,
* насколько далеко приближение может отклоняться от истинной кривой.
**/
class SpiralIterator implements PathIterator {
AffineTransform transform;    // Как преобразовывать
                              // сгенерированные координаты
double flatness;              // Насколько точно приближение
double angle = startAngle;    // Текущий угол
double radius = startRadius;  // Текущий радиус
boolean done = false;         // Закончили?

/** Простой конструктор. Просто сохраняет параметры в полях */
public SpiralIterator(AffineTransform transform, double flatness) {
 this.transform = transform;
 this.flatness = flatness;
}

/**
 * Все объекты PathIterators имеют "правило обхода" ("winding rule"),
 * помогающее указать, что находится внутри окруженной области и что -
 * снаружи. Если залить спираль (не предполагается, что вы будете это
 * делать), возвращаемое здесь "правило обхода" приведет к лучшему
 * результату, чем альтернативное правило WIND_EVEN_ODD
 **/
public int getWindingRule() { return WIND_NON_ZERO; }

/** Возвращаем true, если весь путь пройден */
public boolean isDone() { return done; }

/**
 * Сохраняем координаты текущего отрезка фигуры в заданном массиве
 * и возвращаем тип отрезка. Используем тригонометрические формулы
 * для вычисления координат на основе текущего угла и радиуса.
 * Если это была первая точка, возвращаем отрезок MOVETO, в противном случае
 * возвращаем отрезок LINETO. Также проверяем, не закончена ли работа.
 **/
public int currentSegment(float[] coords) {
 // По заданным углу и радиусу вычисляются координаты точки
 coords[0] = (float)(centerX + radius*Math.cos(angle));
 coords[1] = (float)(centerY - radius*Math.sin(angle));
```

```
 // Если задано преобразование, оно применяется к координатам
 if (transform != null) transform.transform(coords, 0, coords, 0,1);

 // Если достигнут конец спирали, запомним этот факт
 if (angle == endAngle) done = true;

 // Если это первая точка спирали, переместимся к ней
 if (angle == startAngle) return SEG_MOVETO;

 // В противном случае проводим линию от предыдущей точки к этой
 return SEG_LINETO;
}

/** Этот метод такой же, как предыдущий, за исключением того,
    что он использует значения типа double */
public int currentSegment(double[] coords) {
 coords[0] = centerX + radius*Math.cos(angle);
 coords[1] = centerY - radius*Math.sin(angle);
 if (transform != null) transform.transform(coords, 0, coords, 0,1);
 if (angle == endAngle) done = true;
 if (angle == startAngle) return SEG_MOVETO;
 else return SEG_LINETO;
}

/**
 * Переходим к следующему отрезку пути. Вычисляем значения угла
 * и радиуса для следующей точки спирали.
 **/
public void next() {
 if (done) return;

 // Сначала вычисляем, на сколько можно увеличить угол. Это зависит
 // от заданного значения flatness, а также от текущего радиуса. При
 // рисовании окружности с радиусом r (используемой в качестве приближения)
 // можно выдержать точность f, используя шаг угла, задаваемый формулой:
 // a = acos(2*(f/r)*(f/r) - 4*(f/r) + 1)
 // Применим эту формулу для вычисления допустимого увеличения угла
 // для следующего отрезка. Обратите внимание на то, что формула не слишком
 // подходит для маленьких радиусов, поэтому здесь
 // рассматривается особый случай.
 double x = flatness/radius;
 if (Double.isNaN(x) || (x > .1))
 angle += Math.PI/4*angleDirection;
 else {
 double y = 2*x*x - 4*x + 1;
 angle += Math.acos(y)*angleDirection;
 }

 // Проверяем, не проскочили ли мы конец спирали
 if ((angle-endAngle)*angleDirection > 0) angle = endAngle;

 // Теперь, когда мы знаем новый угол, можно применить интерполяцию,
 // чтобы вычислить соответствующий радиус.
 double fractionComplete = (angle-startAngle)/(endAngle-startAngle);
```

```
    radius = startRadius + (endRadius-startRadius)*fractionComplete;
   }
   }
  }
```

Пользовательские классы Stroke

Как мы видели в примере 11.9, класс Stroke преобразует операцию рисования линий в операцию заливки области, принимая объект Shape, очертания которого нужно прорисовать, и возвращая штриховую фигуру, представляющую собой контур исходной фигуры. Поскольку интерфейс Stroke так прост, относительно нетрудно определять собственные классы, реализующие интерфейс Stroke и создающие интересные графические эффекты. Пример 11.15 включает в себя четыре пользовательские реализации Stroke, которые используются вместе с простым объектом BasicStroke для создания изображения, показанного на рис. 11.12.

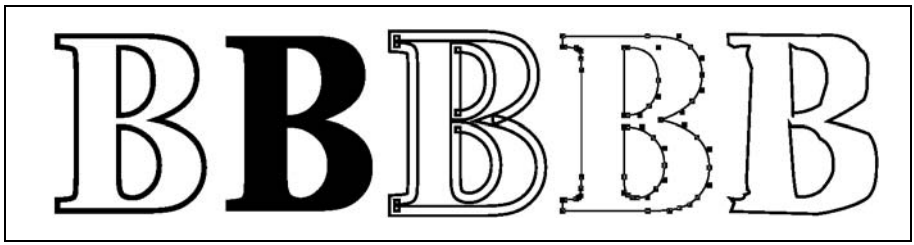


Рис. 11.12. *Специальные эффекты, полученные при помощи пользовательских классов Stroke*

Особое внимание следует уделить реализациям ControlPointsStroke и SloppyStroke. Интересны эти классы тем, что они используют объект PathIterator для разбиения фигур на составляющие их отрезки прямых и кривых (в противоположность тому, что делалось в классе Spiral из примера 11.14). Эти два пользовательских класса Stroke также используют класс GeneralPath пакета java.awt.geom для построения пользовательских фигур из произвольных сегментов прямых и кривых (что показывает, как тесно связаны класс GeneralPath и интерфейс PathIterator).

Пример 11.15. CustomStrokes.java

```
package com.davidflanagan.examples.graphics;
import java.awt.*;
import java.awt.geom.*;
import java.awt.font.*;

/** Демонстрация пользовательских классов Stroke */
public class CustomStrokes implements GraphicsExample {
 static final int WIDTH = 750, HEIGHT = 200;        // Размеры нашего
                                                    // графического примера
```

```
 public String getName() {return "Custom Strokes";} // Из GraphicsExample
 public int getWidth() { return WIDTH; }            // Из GraphicsExample
 public int getHeight() { return HEIGHT; }          // Из GraphicsExample

 // Это различные объекты Stroke, которые мы покажем
 Stroke[] strokes = new Stroke[] {
 new BasicStroke(4.0f),  // Стандартная, заранее определенная штриховая фигура
 new NullStroke(),              // Класс Stroke, не делающий ничего
 new DoubleStroke(8.0f, 2.0f),  // Класс Stroke, обводящий фигуру дважды
 new ControlPointsStroke(2.0f), // Показывает вершины и управляющие точки
 new SloppyStroke(2.0f, 3.0f)   // Искажает фигуру перед созданием
                                // штриховой фигуры
 };

 /** Рисуем пример */
 public void draw(Graphics2D g, Component c) {
 // Получаем фигуру для работы. Здесь мы будем использовать букву B
 Font f = new Font("Serif", Font.BOLD, 200);
 GlyphVector gv = f.createGlyphVector(g.getFontRenderContext(), "B");
 Shape shape = gv.getOutline();

 // Устанавливаем атрибуты рисования и начальную позицию
 g.setColor(Color.black);
 g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
    RenderingHints.VALUE_ANTIALIAS_ON);
 g.translate(10, 175);

 // Рисуем фигуру с каждым штрихом по одному разу
 for(int i = 0; i < strokes.length; i++) {
  g.setStroke(strokes[i]); // устанавливаем штрих
  g.draw(shape);           // рисуем фигуру
  g.translate(140,0);      // перемещаемся вправо
 }
 }
}

/**
 * Эта реализация Stroke не делает ничего. Ее метод createStrokedShape()
 * возвращает неизмененную фигуру. Таким образом, рисование фигуры с этим
 * объектом Stroke – это то же, что заливка фигуры цветом!
 **/
class NullStroke implements Stroke {
 public Shape createStrokedShape(Shape s) { return s; }
}

/**
 * Эта реализация Stroke применяет BasicStroke к фигуре дважды.
 * Рисуя с этим объектом Stroke, вы вместо оконтуривания фигуры
 * оконтуриваете контур фигуры.
 **/
class DoubleStroke implements Stroke {
 BasicStroke stroke1, stroke2;      // два используемых штриха
 public DoubleStroke(float width1, float width2) {
 stroke1 = new BasicStroke(width1); // Аргументы конструкторов задают
```

```
 stroke2 = new BasicStroke(width2); // толщину линии для штриха
 }

 public Shape createStrokedShape(Shape s) {
 // Применяем первый штрих для создания контура заданной фигуры
 Shape outline = stroke1.createStrokedShape(s);
 // Применяем второй штрих для создания контура этого контура.
 // Именно этот контур контура будет залит.
 return stroke2.createStrokedShape(outline);
 }
}

/**
 * Эта реализация Stroke обводит фигуру штрихом, используя тонкую линию,
 * а также показывает концевые и управляющие точки кривых Безье всех отрезков
 * прямых и кривых, составляющих фигуру. Аргумент конструктора radius задает
 * размер маркеров управляющих точек. Обратите внимание
 * на использование PathIterator для разбиения фигуры на сегменты
 * и применение GeneralPath для построения штриховой фигуры.
 **/
class ControlPointsStroke implements Stroke {
 float radius; // такой размер должен быть у маркеров управляющих точек
 public ControlPointsStroke(float radius) { this.radius = radius; }

 public Shape createStrokedShape(Shape shape) {
 // Начнем с обрисовки фигуры штрихом «тонкая линия». Сохраняем получившуюся
 // фигуру в объекте GeneralPath, чтобы иметь возможность дополнять ее.
 GeneralPath strokedShape =
  new GeneralPath(new BasicStroke(1.0f).createStrokedShape(shape));

 // Применяем объект PathIterator для обхода всех прямых и криволинейных
 // отрезков фигуры. Для каждого из них отмечаем концевые и управляющие
 // точки (если таковые имеются), добавляя к GeneralPath прямоугольники
 float[] coords = new float[6];
 for(PathIterator i=shape.getPathIterator(null); !i.isDone();i.next()) {
  int type = i.currentSegment(coords);
  Shape s = null, s2 = null, s3 = null;
  switch(type) {
  case PathIterator.SEG_CUBICTO:
  markPoint(strokedShape, coords[4], coords[5]); // проваливаемся
  case PathIterator.SEG_QUADTO:
  markPoint(strokedShape, coords[2], coords[3]); // проваливаемся
  case PathIterator.SEG_MOVETO:
  case PathIterator.SEG_LINETO:
  markPoint(strokedShape, coords[0], coords[1]); // проваливаемся
  case PathIterator.SEG_CLOSE:
  break;
  }
 }

 return strokedShape;
 }
 /** Добавляем маленький квадратик с центром в точке (x,y) к заданному пути */
```

```
 void markPoint(GeneralPath path, float x, float y) {
 path.moveTo(x-radius, y-radius); // Начинаем новый подпуть
 path.lineTo(x+radius, y-radius); // Добавляем к нему отрезок
 path.lineTo(x+radius, y+radius); // Добавляем второй отрезок
 path.lineTo(x-radius, y+radius); // Добавляем третий
 path.closePath();   // Возвращаемся к последнему положению moveTo
 }
}

/**
 * Эта реализация Stroke случайным образом сдвигает отрезки прямых и кривых,
 * составляющих Shape, а затем обводит эту искаженную фигуру.
 * Она использует PathIterator для обхода фигуры Shape в цикле и GeneralPath -
 * для построения преобразованной фигуры. Наконец, она использует BasicStroke
 * для обрисовки преобразованной фигуры штрихом. В результате получается
 * "криво" нарисованная фигура.
 **/
class SloppyStroke implements Stroke {
 BasicStroke stroke;
 float sloppiness;
 public SloppyStroke(float width, float sloppiness) {
 this.stroke = new BasicStroke(width); // Применяется для обрисовки
                                       // преобразованной фигуры штрихом
 this.sloppiness = sloppiness;         // Насколько криво будем рисовать?
 }

 public Shape createStrokedShape(Shape shape) {
 GeneralPath newshape = new GeneralPath(); // Начинаем с пустой фигуры

 // Обходим заданную фигуру, сдвигаем ее координаты
 // и используем их для построения новой фигуры.
 float[] coords = new float[6];
 for(PathIterator i=shape.getPathIterator(null); !i.isDone();i.next()) {
  int type = i.currentSegment(coords);
  switch(type) {
  case PathIterator.SEG_MOVETO:
  perturb(coords, 2);
  newshape.moveTo(coords[0], coords[1]);
  break;
  case PathIterator.SEG_LINETO:
  perturb(coords, 2);
  newshape.lineTo(coords[0], coords[1]);
  break;
  case PathIterator.SEG_QUADTO:
  perturb(coords, 4);
  newshape.quadTo(coords[0], coords[1], coords[2], coords[3]);
  break;
  case PathIterator.SEG_CUBICTO:
  perturb(coords, 6);
  newshape.curveTo(coords[0], coords[1], coords[2], coords[3],
     coords[4], coords[5]);
  break;
```

```
    case PathIterator.SEG_CLOSE:
    newshape.closePath();
    break;
    }
  }

  // Наконец, обводим искаженную фигуру и возвращаем результат
  return stroke.createStrokedShape(newshape);
  }

  // Случайным образом изменяем заданное множество координат на величину,
  // заданную в поле sloppiness.
  void perturb(float[] coords, int numCoords) {
  for(int i = 0; i < numCoords; i++)
    coords[i] += (float)((Math.random()*2-1.0)*sloppiness);
  }
}
```

Пользовательские классы Paint

На рис. 11.8 было показано много разных способов заливки фигур; там есть большая буква A, залитая сложной шаблонной текстурой, определенной классом `GenericPaint`. Пример 11.16 показывает реализацию этого класса. Перед тем как углубиться в приведенный здесь код, вам, пожалуй, будет полезно еще раз взглянуть на пример 11.10, чтобы посмотреть, как там применялся класс `GenericPaint`.

Класс `GenericPaint` сам по себе довольно прост: он определяет, во-первых, абстрактные методы вычисления цвета, реализованные в подклассах, и, во-вторых, метод `createContext()`, возвращающий `PaintContext`. Реализация `PaintContext` выполняет самую сложную часть работы. В ней используется довольно-таки низкоуровневый подход, поэтому пусть вас не обескураживает, если вы не во всем разберетесь. Этот код во всяком случае должен дать вам общее представление о том, как выполняется рисование в Java 2D.

Пример 11.16. GenericPaint.java

```
package com.davidflanagan.examples.graphics;
import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;

/**
 * Это абстрактная реализация интерфейса Paint, вычисляющая цвет
 * для каждой точки будущего изображения посредством передачи координат точки
 * абстрактным методам computeRed(), computeGreen(), computeBlue()
 * и computeAlpha(). Подклассы должны реализовать эти методы
 * в соответствии с требуемым изображением. Заметим, что хотя
 * этот класс обеспечивает большую гибкость, он не слишком эффективен.
 **/
```

```
public abstract class GenericPaint implements Paint {
 /** Это главный метод в Paint; все, что он делает, -
     возвращает объект PaintContext */
 public PaintContext createContext(ColorModel cm,
     Rectangle deviceBounds,
     Rectangle2D userBounds,
     AffineTransform xform,
     RenderingHints hints) {
 return new GenericPaintContext(xform);
 }

 /** Этот метод позволяет применять прозрачное рисование  */
 public int getTransparency() { return TRANSLUCENT; }

 /**
 * Эти четыре метода возвращают красную, зеленую и синюю составляющие цвета
 * и уровень прозрачности Alpha для пиксела с заданными координатами
 * в пространстве пользователя. Для всех методов возвращаемое
 * значение должно находиться в промежутке от 0 до 255.
 **/
 public abstract int computeRed(double x, double y);
 public abstract int computeGreen(double x, double y);
 public abstract int computeBlue(double x, double y);
 public abstract int computeAlpha(double x, double y);

 /**
 * Класс, реализующий PaintContext и выполняющий всю художественную работу
 **/
 class GenericPaintContext implements PaintContext {
 ColorModel model;                          // Цветовая модель
 Point2D origin, unitVectorX, unitVectorY; // Для перевода из пространства
                               // устройства в пространство пользователя

 public GenericPaintContext(AffineTransform userToDevice) {
  // Наша цветовая модель упаковывает ARGB-значения в одно целое число
  model = new DirectColorModel(32, 0x00ff0000,0x0000ff00,
      0x000000ff, 0xff000000);
  // Заданное преобразование переводит пользовательские пикселы в пикселы
  // устройства. Нам нужно создать обратное преобразование, чтобы иметь
  // возможность вычислить пользовательские координаты
  // для каждого пиксела устройства
  try {
  AffineTransform deviceToUser = userToDevice.createInverse();
  origin = deviceToUser.transform(new Point(0,0), null);
  unitVectorX = deviceToUser.deltaTransform(new Point(1,0),null);
  unitVectorY = deviceToUser.deltaTransform(new Point(0,1),null);
  }
  catch (NoninvertibleTransformException e) {
  // Если преобразование необратимо, просто используем
  // пространство устройства
  origin = new Point(0,0);
  unitVectorX = new Point(1,0);
  unitVectorY = new Point(0, 1);
```

```
 }
}

/** Возвращаем цветовую модель, используемую этой реализацией Paint */
public ColorModel getColorModel() { return model; }

/**
 * Это главный метод в PaintContext. Он должен возвращать объект Raster,
 * содержащий данные о заливке для заданного прямоугольника. Он создает
 * растр заданного размера, а затем в цикле пробегает пикселы устройства,
 * для каждого из которых он переводит координаты в пространство
 * пользователя, а затем вызывает методы computeRed(), computeGreen()
 * и computeBlue() для получения цвета для пиксела устройства.
 **/
public Raster getRaster(int x, int y, int w, int h) {
 WritableRaster raster = model.createCompatibleWritableRaster(w,h);
 int[] colorComponents = new int[4];
 for(int j = 0; j < h; j++) { // Цикл по строкам растра
 int deviceY = y + j;
 for(int i = 0; i < w; i++) { // Цикл по столбцам
  int deviceX = x + i;
  // Переводим координаты устройства в пользовательское пространство
  double userX = origin.getX() +
  deviceX * unitVectorX.getX() +
  deviceY * unitVectorY.getX();
  double userY = origin.getY() +
  deviceX * unitVectorX.getY() +
  deviceY * unitVectorY.getY();
  // Вычисляем цветовые составляющие для пиксела
  colorComponents[0] = computeRed(userX, userY);
  colorComponents[1] = computeGreen(userX, userY);
  colorComponents[2] = computeBlue(userX, userY);
  colorComponents[3] = computeAlpha(userX, userY);
  // Устанавливаем цвет пиксела
  raster.setPixel(i, j, colorComponents);
 }
 }
 return raster;
}

/** Этот метод вызывается, когда PaintContext уже не нужен. */
public void dispose() {}
}
}
```

Сложная анимация

В примере 11.4 мы видели простую технику анимации, которая, к сожалению, страдала мерцанием. Пример 11.17 – это программа, выполняющая более насыщенную графикой анимацию, которая, к тому же, не мерцает, поскольку в ней применяется прием, известный как

двойная буферизация: каждый кадр анимации рисуется вне экрана, а затем целиком копируется на экран. И производительность этого примера выше, поскольку ему приходится перерисовывать относительно небольшую часть экрана – лишь то, что нужно изменить.

Другая интересная особенность этого примера состоит в использовании класса `javax.swing.Timer` для вызова с заданными интервалами метода `actionPerformed()` заданного объекта `ActionListener`. Класс `Timer` применяется здесь для того, чтобы исключить необходимость создания потока `Thread`. (Обратите внимание, что Java 1.3 включает в себя класс `java.util.Timer`, который похож на `javax.swing.Timer`, но все же отличается от него.)

Пример 11.17. Hypnosis.java

```
package com.davidflanagan.examples.graphics;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;
import javax.swing.Timer; // Импортируем явно из-за java.util.Timer

/**
 * Компонент Swing реализует очень плавную анимацию спирали.
 **/
public class Hypnosis extends JComponent implements ActionListener {
 double x, y;           // Центр спирали
 double r1, r2;         // Внутренний и внешний радиусы спирали
 double a1, a2;         // Начальный и конечный углы спирали
 double deltaA;         // Величина изменения угла между кадрами
 double deltaX, deltaY; // Траектория центра
 float linewidth;       // Толщина линий
 Timer timer;           // Объект, управляющий сменой кадров
 BufferedImage buffer;  // Изображение, используемое для двойной буферизации
 Graphics2D osg;        // Объект Graphics2D для рисования в буфер

 public Hypnosis(double x, double y, double r1, double r2,
                 double a1, double a2, float linewidth, int delay,
                 double deltaA, double deltaX, double deltaY)
 {
    this.x = x; this.y = y;
    this.r1 = r1; this.r2 = r2;
    this.a1 = a1; this.a2 = a2;
    this.linewidth = linewidth;
    this.deltaA = deltaA;
    this.deltaX = deltaX;
    this.deltaY = deltaY;

 // Настраиваем таймер на вызов actionPerformed() каждые delay миллисекунд
 timer = new Timer(delay, this);

 // Создаем буфер для двойной буферизации
 buffer = new BufferedImage((int)(2*r2+linewidth),
                            (int)(2*r2+linewidth), BufferedImage.TYPE_INT_RGB);
```

```
// Создаем объект Graphics для буфера, устанавливаем толщину линии
// и требуем сглаживания при рисовании
osg = buffer.createGraphics();
osg.setStroke(new BasicStroke(linewidth, BasicStroke.CAP_ROUND,
    BasicStroke.JOIN_ROUND));
osg.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
   RenderingHints.VALUE_ANTIALIAS_ON);
}

// Запускаем и останавливаем анимацию, запуская и останавливая таймер
public void start() { timer.start(); }
public void stop() { timer.stop(); }

/**
* Swing вызывает этот метод, требуя, чтобы компонент перерисовал себя.
* Этот метод применяет двойную буферизацию, чтобы сделать анимацию более
* плавной. Swing выполняет двойную буферизацию автоматически, так что на
* самом деле большой разницы не будет, но эту технику важно понять.
**/
public void paintComponent(Graphics g) {
// Очищаем фон внеэкранного изображения
osg.setColor(getBackground());
osg.fillRect(0, 0, buffer.getWidth(), buffer.getHeight());

// Теперь рисуем черную спираль на внеэкранном изображении
osg.setColor(Color.black);
osg.draw(new Spiral(r2+linewidth/2, r2+linewidth/2, r1, a1, r2, a2));

// Теперь копируем внеэкранное изображение на экран
g.drawImage(buffer, (int)(x-r2), (int)(y-r2), this);
}

/**
* Этот метод реализует интерфейс ActionListener. Наш объект Timer
* вызывает этот метод периодически. Он обновляет положение и углы
* спирали и требует перерисовывания. Вместо перерисовывания всего
* компонента этот метод требует перерисовать только область изменений.
**/
public void actionPerformed(ActionEvent e) {
// Просим перерисовать старый ограничивающий прямоугольник спирали.
// Ничего другого нарисовано не было, поэтому нечего и перерисовывать
repaint((int)(x-r2-linewidth), (int)(y-r2-linewidth),
 (int)(2*(r2+linewidth)), (int)(2*(r2+linewidth)));

// Теперь анимация: обновляем положение и углы спирали

// "Отскок" при столкновении с краем
Rectangle bounds = getBounds();
if ((x - r2 + deltaX < 0) || (x + r2 + deltaX > bounds.width))
    deltaX = -deltaX;
if ((y - r2 + deltaY < 0) || (y + r2 + deltaY > bounds.height))
    deltaY = -deltaY;
```

```
// Перемещаем центр спирали
 x += deltaX;
 y += deltaY;

 // Увеличиваем начальный и конечный углы;
 a1 += deltaA;
 a2 += deltaA;
 if (a1 > 2*Math.PI) { // Не позволяем им становиться слишком большими
  a1 -= 2*Math.PI;
  a2 -= 2*Math.PI;
 }

 // Теперь просим перерисовать новый ограничивающий прямоугольник спирали.
 // Этот прямоугольник пересечем с прямоугольником, перерисовываемым выше,
 // и перерисовываться будет только их объединение
 repaint((int)(x-r2-linewidth), (int)(y-r2-linewidth),
  (int)(2*(r2+linewidth)), (int)(2*(r2+linewidth)));
 }

 /** Даем Swing команду не производить для нас двойную буферизацию,
     поскольку мы ее делаем сами */
 public boolean isDoubleBuffered() { return false; }

 /** Это метод main() для тестирования компонента */
 public static void main(String[] args) {
 JFrame f = new JFrame("Hypnosis");
 Hypnosis h = new Hypnosis(200, 200, 10, 100, 0, 11*Math.PI, 7, 100,
     2*Math.PI/30, 3, 5);
 f.getContentPane().add(h, BorderLayout.CENTER);
 f.setSize(400, 400);
 f.show();
 h.start();
 }
}
```

Отображение графических примеров

Пример 11.18 показывает класс `GraphicsExampleFrame`**, который мы использовали на протяжении этой главы для отображения реализаций** `GraphicsExample`**. Эта программа демонстрирует, главным образом, Swing и Printing API и включена сюда для полноты. Метод** `paintComponent()` **внутреннего класса** `GraphicsExamplePane` **находится там, где вызывается метод** `draw()` **каждого из объектов** `GraphicsExample`**. Хотя** `paintComponent()` **объявлен как получающий в качестве аргумента объект** `Graphics`**, начиная с Java 1.2 ему всегда передается объект** `Graphics2D`**, который может быть безопасно приведен к этому типу.**

Пример 11.18. GraphicsExampleFrame.java

```
package com.davidflanagan.examples.graphics;
import java.awt.*;
import java.awt.event.*;
```

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.print.*;

/**
 * Этот класс отображает один или несколько объектов GraphicsExample
 * при помощи Swing-компонентов JFrame и JTabbedPane
 */
public class GraphicsExampleFrame extends JFrame {
 public GraphicsExampleFrame(final GraphicsExample[] examples) {
 super("GraphicsExampleFrame");

 Container cpane = getContentPane(); // Задаем окно
 cpane.setLayout(new BorderLayout());
 final JTabbedPane tpane = new JTabbedPane(); // И панель со вкладками
 cpane.add(tpane, BorderLayout.CENTER);

 // Добавляем полосу меню
 JMenuBar menubar = new JMenuBar();        // Создаем полосу меню
 this.setJMenuBar(menubar);                // Помещаем ее в окно
 JMenu filemenu = new JMenu("File");       // Создаем меню File
 menubar.add(filemenu);                    // Помещаем его в полосу меню
 JMenuItem print = new JMenuItem("Print"); // Создаем элемент Print
 filemenu.add(print);                      // Помещаем его в меню
 JMenuItem quit = new JMenuItem("Quit");   // Создаем элемент Quit
 filemenu.add(quit);                       // Помещаем его в меню

 // Сообщаем элементу меню Print, что ему делать, когда его выберут
 print.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
   // Получаем отображаемый в настоящий момент графический пример и вызываем
   // метод print (определенный ниже)
   print(examples[tpane.getSelectedIndex()]);
  }
  });

 // Сообщаем элементу меню Quit, что ему делать, когда его выберут
 quit.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) { System.exit(0); }
  });

 // Здесь же обрабатываем закрытие окна
 this.addWindowListener(new WindowAdapter() {
  public void windowClosing(WindowEvent e) { System.exit(0); }
  });

 // Помещаем все объекты графических примеров на панель со вкладками
 for(int i = 0; i < examples.length; i++) {
  GraphicsExample e = examples[i];
  tpane.addTab(e.getName(), new GraphicsExamplePane(e));
 }
 }

 /**
 * Этот внутренний класс представляет собой пользовательский компонент Swing,
```

```
* отображающий объект GraphicsExample.
*/
public class GraphicsExamplePane extends JComponent {
GraphicsExample example; // Отображаемый графический пример
Dimension size;          // Сколько места ему требуется

public GraphicsExamplePane(GraphicsExample example) {
 this.example = example;
 size = new Dimension(example.getWidth(), example.getHeight());
}

/** Рисуем компонент и графический пример, который он содержит */
public void paintComponent(Graphics g) {
 g.setColor(Color.white);                    // Цвет фона
 g.fillRect(0, 0, size.width, size.height); // устанавливаем белым.
 g.setColor(Color.black);            // Устанавливаем принимаемый
                                     // по умолчанию цвет рисунка
 example.draw((Graphics2D) g, this); // Командуем примеру перерисовать себя
}

// Эти методы указывают, насколько велик должен быть компонент
public Dimension getPreferredSize() { return size; }
public Dimension getMinimumSize() { return size; }
}

/** Этот метод вызывается элементом меню Print */
public void print(final GraphicsExample example) {
// Начинаем с получения задания принтеру (printer job) для выполнения печати
PrinterJob job = PrinterJob.getPrinterJob();
// Помещаем пример в объект-оболочку Printable (определенный ниже)
// и сообщаем заданию PrinterJob, что мы хотим его напечатать
job.setPrintable(new PrintableExample(example));

// Выводим для пользователя диалоговое окно печати
if (job.printDialog()) {
 // Если пользователь не отменяет задание, даем заданию команду начать печать
 try {
 job.print();
 }
 catch(PrinterException e) {
 System.out.println("Невозможно напечатать: " + e.getMessage());
 }
}
}

/**
* Этот внутренний класс реализует интерфейс Printable для целей
* печати объекта GraphicsExample.
**/
class PrintableExample implements Printable {
GraphicsExample example; // Печатаемый графический пример

// Конструктор. Просто запоминает пример
public PrintableExample(GraphicsExample example) {
```

```
 this.example = example;
}

/**
 * Этот метод вызывается задачей PrinterJob для печати графического примера
 **/
public int print(Graphics g, PageFormat pf, int pageIndex) {
 // Сообщаем PrinterJob, что имеется только одно задание
 if (pageIndex != 0) return NO_SUCH_PAGE;

 // PrinterJob предоставляет нам объект Graphics для рисования.
 // Все, что нарисовано при помощи этого объекта, будет послано на принтер.
 // Объект Graphics может быть безопасно приведен к Graphics2D.
 Graphics2D g2 = (Graphics2D)g;

 // Отступаем на величину верхнего и левого полей.
 g2.translate(pf.getImageableX(), pf.getImageableY());

 // Вычисляем размер области, доступной для печати, и размер примера.
 double pageWidth = pf.getImageableWidth();
 double pageHeight = pf.getImageableHeight();
 double exampleWidth = example.getWidth();
 double exampleHeight = example.getHeight();

 // При необходимости масштабируем пример
 double scalex = 1.0, scaley = 1.0;
 if (exampleWidth > pageWidth) scalex = pageWidth/exampleWidth;
 if (exampleHeight > pageHeight) scaley = pageHeight/exampleHeight;
 double scalefactor = Math.min(scalex, scaley);
 if (scalefactor != 1) g2.scale(scalefactor, scalefactor);

 // Наконец, вызываем метод draw() примера, передавая принтеру
 // объект Graphics2D
 example.draw(g2, GraphicsExampleFrame.this);

 // Сооощаем PrinterJob, что страница успешно выведена
 return PAGE_EXISTS;
}
}

/**
* Главная программа. Применяем отражение Java для загрузки заданных классов
* GraphicsExample и создания их объектов, а затем создаем
* GraphicsExampleFrame, чтобы их отобразить.
**/
public static void main(String[] args) {
GraphicsExample[] examples = new GraphicsExample[args.length];

// Цикл по аргументам командной строки
for(int i=0; i < args.length; i++) {
 // Имя класса запрашиваемого примера
 String classname = args[i];

 // Если никакой пакет не задан, предполагаем, что класс
 // находится в этом пакете
```

```
        if (classname.indexOf('.') == -1)
        classname = "com.davidflanagan.examples.graphics."+args[i];

        // Пытаемся создать объект указанного класса GraphicsExample
        try {
        Class exampleClass = Class.forName(classname);
        examples[i] = (GraphicsExample) exampleClass.newInstance();
        }
        catch (ClassNotFoundException e) { // неизвестный класс
        System.err.println("Невозможно найти пример: " + classname);
        System.exit(1);
        }
        catch (ClassCastException e) { // неправильный тип класса
        System.err.println("Класс " + classname +
           " не является классом GraphicsExample");
        System.exit(1);
        }
        catch (Exception e) { // у класса нет открытого конструктора
        // Перехватываем InstantiationException и IllegalAccessException
        System.err.println("Невозможно создать экземпляр примера: " +
           classname);
        System.exit(1);
        }
      }

      // Теперь создаем окно для отображения в нем примеров и делаем его видимым
      GraphicsExampleFrame f = new GraphicsExampleFrame(examples);
      f.pack();
      f.setVisible(true);
      }
    }
```

Упражнения

11-1. Пример 11.1 содержит метод `centerText()`, центрирующий одну или две строки в прямоугольнике. Напишите измененную версию этого метода, которая позиционирует одну стоку текта в соответствии с двумя новыми параметрами метода. Один параметр должен задавать положение по горизонтали: выравнивание по центру, по левому или по правому краю. Другой параметр должен задавать положение по вертикали: у верхнего края прямоугольника, посередине или у нижнего края прямоугольника. Вы можете использовать класс `FontMetrics`, как это делалось в примере 11.1, или Java 2D-метод `getStringBounds()` класса `Font`. Напишите класс `GraphicsExample`, демонстрирующий все девять возможных типов позиционирования, поддерживаемых вашим методом.

11-2. Используйте показанные в этой главе приемы анимации, написав апплет, выводящий на экран бегущую строку. Прокручиваемый текст и скорость прокручивания должны считываться из тегов апплета <PARAM> в файле HTML.

11-3. Поэкспериментируйте с графическими возможностями классов Graphics и Graphics2D и напишите апплет или приложение, отображающие какую-нибудь интересную динамическую графику. Возможно, вас вдохновит одна из программ экранных заставок, которых много в продаже. Например, вы можете рисовать на экране залитые прямоугольники, используя случайные положения, размеры и цвета. (Способы генерирования случайных чисел см. в Math.random() и java.util.Random.) Не стесняйтесь использовать любой из определенных в этой главе классов Shape, Stroke или Paint. Проявите фантазию!

11-4. Мощь графики не в последнюю очередь обусловлена тем, что она позволяет нам визуализировать абстрактные модели. Вернемся к примеру 1.15. Эта программа вычисляет простые числа при помощи «решета Эратосфена», алгоритма, по которому для нахождения простого числа из ряда натуральных чисел удаляются все кратные меньшим простым числам. В процессе выполнения этого алгоритма мы поддерживали массив, показывавший, является ли число простым. Напишите графическую версию программы Sieve. Она должна отображать массив в виде двумерной (2D) матрицы; ячейки матрицы, представляющие простые числа, должны отображаться одним цветом; составные числа – другим. Напишите программу в виде самостоятельного приложения или (если вам так больше нравится) как реализацию GraphicsExample.

И еще расширим вашу программу. При вычислении простых чисел сохраняйте в массиве не значения типа boolean, отличающие простые числа от составных, а значения типа int, которые для составных чисел будут содержать то простое число, которое отсеяло данное число. После вычисления множества простых чисел отобразите содержимое вашего массива в виде прямоугольной сетки, используя различные цвета для отображения кратных каждому из простых. Появляющиеся образцы лучше всего будут выглядеть в квадратной сетке с простым числом ячеек по ширине и по высоте. Если, например, вычисляются все простые числа, не превосходящие 361, можно графически отобразить результаты вычисления в сетке 19×19. Изучите полученные картинки. Помогают ли они вам проникнуть в механизм алгоритма? Помогли ли они понять, почему его называют «решетом»?

11-5. Пример 11.10 показывает, как можно использовать элементы шрифта в качестве объектов Shape и как преобразовывать эти объекты Shape при помощи объекта AffineTransform. Напишите пользовательский объект типа Stroke, который инициализирует-

ся объектом `Font` и строкой текста. Когда этот класс `Stroke` применяется для рисования фигуры, он должен отображать заданный текст вдоль контура заданной фигуры с использованием заданного шрифта. Текст должен прилегать к контуру фигуры так близко, как только возможно, то есть каждый символ шрифта должен быть повернут, размещен и нарисован независимо от других символов. Используйте двухаргументную версию метода `Shape.getPathIterator()` для получения `FlatteningPathIterator`, аппроксимирующего фигуру отрезками прямых. Вам понадобятся геометрические формулы для вычисления наклона каждого отрезка и тригонометрические – для преобразования наклона в угол поворота каждого символа шрифта (вам сможет помочь `Math.atan2()`). Напишите класс `GraphicsExample`, отображающий некий художественный текст для демонстрации вашего объекта `Stroke`.

Глава 12

Печать

Объекты `Graphics` и `Graphics2D` представляют собой «поверхность рисования». В главе 11 мы видели примеры использования в качестве поверхности рисования как экрана, так и внеэкранного буфера. Для выполнения печати в Java просто нужно получить объект `Graphics`, который использует в качестве поверхности рисования принтер. Если у вас есть объект `Graphics`, вы можете печатать текст и выводить графику на принтер, как вы делаете это с экраном. Единственная хитрость относится к получению объекта `Graphics`, связанного с принтером. В Java 1.1 для этого был введен один программный интерфейс, а затем в Java 1.2 был представлен новый, отличный от него API.[1] Эта глава содержит примеры печати компонентов и многостраничных документов, использующие оба API.

Печать с помощью API Java 1.1

В примере 12.1 мы снова вернемся к приложению рисования изображений из главы 10 «Графические интерфейсы пользователя (GUI)». Мы рассмотрим графический компонент, позволяющий пользователю рисовать мышью. Этот пример запоминает координаты выделенного фрагмента и при нажатии пользователем кнопки **Print** распечатывает его.

В примере используются библиотека AWT (не Swing) и программный интерфейс печати Java 1.1. В методе `printScribble()` показаны шаги, необходимые при использовании этого интерфейса. Непосредственно

[1] Затем в Java 1.3 были добавлены расширения Java 1.1 API в виде классов `JobAttributes` и `PageAttributes`. И ожидается, что Java 1.4 расширит интерфейс печати Java 1.2.

печать выполняется в методе `paint()`**, в котором также выполняется внеэкранное рисование.**

Пример 12.1. ScribblePrinter1.java

```
/**
* "Приложение для рисования", запоминающее фрагмент изображения
* и позволяющее пользователю распечатать его. Оно использует интерфейс AWT
* для отображения и интерфейс печати Java 1.1.
**/
public class ScribblePrinter1 extends Panel {
    // Координаты последнего щелчка мышью
    private short last_x = 0, last_y = 0;
    // Сохраняем фрагмент
    private Vector lines = new Vector(256,256);
    // Сохраняем пользовательские настройки
    private Properties printprefs = new Properties();
    private Frame frame;

    public ScribblePrinter1(Frame frame) {
        // Запоминаем рамку: она потребуется для создания объекта PrintJob
        this.frame = frame;

        // Регистрируем типы интересующих нас событий.
        enableEvents(AWTEvent.MOUSE_EVENT_MASK |
                     AWTEvent.MOUSE_MOTION_EVENT_MASK);

        // Помещаем кнопку «Print» в ее контейнер
        // и выполняем печать при ее нажатии
        Button b = new Button("Print");
        b.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            printScribble(); }
        });
        this.setLayout(new FlowLayout(FlowLayout.RIGHT, 5, 5));
        this.add(b);
    }

    /** Перерисовываем (или печатаем) фрагмент,
        используя сохраненные координаты линий */
    public void paint(Graphics g) {
        for(int i = 0; i < lines.size(); i++) {
            Line l = (Line)lines.elementAt(i);
            g.drawLine(l.x1, l.y1, l.x2, l.y2);
        }
    }

    /** Распечатываем фрагмент */
    public void printScribble() {
        // Получаем объект PrintJob
        Toolkit toolkit = this.getToolkit();
        PrintJob job = toolkit.getPrintJob(frame, "ScribblePrinter1",
                                                  printprefs);
```

```
        // Если пользователь нажал кнопку "Cancel" в окне диалога,
        // печать не выполняется
        if (job == null) return;

        // Получаем объект Graphics, который мы используем
        // для рисования на принтере
        Graphics g = job.getGraphics();

        // Устанавливаем верхний и левый отступы для зоны вывода.
        // Иначе она будет сдвинута в левый верхний угол страницы.
        g.translate(100, 100);

        // Рисуем рамку вокруг зоны вывода.
        Dimension size = this.getSize();
        g.drawRect(-1, -1, size.width+2, size.height+2);

        // Устанавливаем область отсечки, чтобы наш рисунок не выводился
        // за пределами отмеченной рамки. При печати вне экрана
        // это происходит автоматически, но для бумаги это не так.
        g.setClip(0, 0, size.width, size.height);

        // Печатаем данный компонент и все содержащиеся в нем компоненты.
        // Здесь будет вызван метод paint(),а также будут выводиться кнопки.
        // Если вы не хотите показывать кнопки, используйте
        // вместо printAll() метод print().
        this.printAll(g);

        //Завершение.
        g.dispose();       // Завершаем текущую страницу
        job.end();         // Заканчиваем печать
    }

    /** Этот метод вызывается, когда пользователь щелкнет мышью
        для начала рисования линии */
    public void processMouseEvent(MouseEvent e) {
        if (e.getID() == MouseEvent.MOUSE_PRESSED) {
            // запоминаем координаты нажатия
            last_x = (short)e.getX();
            last_y = (short)e.getY();
        }
        else super.processMouseEvent(e);
    }

    /** Этот метод вызывается, когда пользователь перемещает мышь:
        он выполняет рисование линии */
    public void processMouseMotionEvent(MouseEvent e) {
        if (e.getID() == MouseEvent.MOUSE_DRAGGED) {
            Graphics g = getGraphics();
            // Рисуем линию
            g.drawLine(last_x, last_y, e.getX(), e.getY());
            // и запоминаем ее
            lines.addElement(new Line(last_x, last_y,
                     (short) e.getX(), (short)e.getY()));
            last_x = (short) e.getX();
```

```
            last_y = (short) e.getY();
        }
        else super.processMouseMotionEvent(e);
    }

    /** Метод main(). Создает объект ScribblePrinter1 и завершается */
    public static void main(String[] args) {
        Frame frame = new Frame("ScribblePrinter1");
        ScribblePrinter1 s = new ScribblePrinter1(frame);
        frame.add(s, BorderLayout.CENTER);
        frame.setSize(400, 400);
        frame.show();
    }

    /**
     * В этом внутреннем классе запоминаются координаты одной линии рисунка.
     **/
    class Line {
        public short x1, y1, x2, y2;
        public Line(short x1, short y1, short x2, short y2) {
            this.x1 = x1; this.y1 = y1;
            this.x2 = x2; this.y2 = y2;
        }
    }
}
```

Печать с помощью API Java 1.2

В примере 12.2 выполняется то же самое, что в примере 12.1, но он изменен так, чтобы использовать библиотеки Swing, Java 2D и интерфейс печати Java 1.2. Java 1.2 API находится в пакете `java.awt.print`**. Обратите внимание на то, что в нем используется класс** `java.awt.print.PrinterJob`**, а не класс** `java.awt.PrintJob` **интерфейса Java 1.1. В этом примере наш класс реализует интерфейс** `java.awt.print.Printable`**. Это означает, что в нем определен метод** `print()`**, который позволяет методу** `PrinterJob.printScribble()` **распечатать его, просто получив объект** `PrinterJob` **и дав ему указание напечататься. Все остальное метод** `print()` **сделает сам. В этом примере, как и в предыдущем, для непосредственного выполнения печати используется собственный метод компонента, в данном случае – `paintComponent()`. Вы можете найти еще один пример использования интерфейса печати Java 1.2 в примере 11.18.**

Пример 12.2. ScribblePrinter2.java

```
package com.davidflanagan.examples.print;
import java.awt.*;
import java.awt.event.*;
import java.awt.print.*;
import java.awt.geom.*;
import javax.swing.*;
```

```
import java.util.*;

/**
 * "Приложение для рисования", запоминающее рисунок и позволяющее
 * пользователю распечатывать его. В нем используются интерфейс Swing
 * для отображения и интерфейс печати Java 1.2. В нем также используются
 * средства Java2D для рисования и представления рисунка.
 **/
public class ScribblePrinter2 extends JComponent implements Printable {
    // Рисуем толстыми линиями
    Stroke linestyle = new BasicStroke(3.0f);
    // Место для хранения рисунка
    GeneralPath scribble = new GeneralPath();

    public ScribblePrinter2() {
        // Регистрируем типы событий, которые нам нужны
        // для рисования
        enableEvents(AWTEvent.MOUSE_EVENT_MASK |
                     AWTEvent.MOUSE_MOTION_EVENT_MASK);

        // Добавляем кнопку «Print» в ее область и отвечаем
        // на ее нажатие печатью
        JButton b = new JButton("Print");
        b.addActionListener(new ActionListener() {
               public void actionPerformed(ActionEvent e) { printScribble();
}
            });
        this.setLayout(new FlowLayout(FlowLayout.LEFT, 5, 5));
        this.add(b);
    }

    /** Перерисовываем (или печатаем) рисунок на основе запомненных линий */
    public void paintComponent(Graphics g) {
        // Разрешаем базовому классу нарисовать себя
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D) g;
        // Указываем толстые линии
        g2.setStroke(linestyle);
        g2.draw(scribble);          // Выводим рисунок
    }

    /**
     * Распечатываем рисунок. Этот метод вызывается
     * при нажатии кнопки "Print"; он не является частью интерфейса Printable
     **/
    public void printScribble() {
        // получаем объект java.awt.print.PrinterJob
        // (не java.awt.PrintJob)
        PrinterJob job = PrinterJob.getPrinterJob();

        // Даем объекту PrinterJob команду напечатать нас
        // (т.к. мы реализуем интерфейс Printable), используя параметры
        // страницы, принятые по умолчанию.
        job.setPrintable(this, job.defaultPage());
```

```
        // Выводим диалог печати, позволяющий пользователю
        // установить настройки. Этот метод возвращает false,
        // если пользователь отменил запрос на печать
        if (job.printDialog()) {
        // Если не было отмены, начинаем печать! Вызывается метод print(),
        // определенный в интерфейсе Printable.
        try { job.print(); }
        catch (PrinterException e) {System.err.println(e);}
        }
    }

    /**
     * Этот метод определен в интерфейсе Printable. Он распечатывает рисунок
     * в указанном объекте Graphics с учетом размера бумаги и полей,
     * заданных объектом PageFormat. Если указанный номер страницы
     * не равен 0, возвращается код, говорящий о том, что печать завершена.
     * Метод должен быть готов к тому, что при каждом запросе
     * на печать он будет вызываться несколько раз.
     **/
    public int print(Graphics g, PageFormat format, int pagenum) {
        // Наша длина - только одна страница
        if (pagenum > 0) return Printable.NO_SUCH_PAGE;

        // Интерфейс печати Java 1.2 передает нам объект Graphics,
        // но мы всегда можем привести его к объекту Graphics2D
        Graphics2D g2 = (Graphics2D) g;

        // Преобразуем для подгонки к указанным верхнему и левому полям.
        g2.translate(format.getImageableX(), format.getImageableY());

        // Выясняем величину рисунка и страницы (за вычетом полей).
        // Размер рисунка
        Dimension size = this.getSize();
        // Ширина страницы
        double pageWidth = format.getImageableWidth();
        // Высота страницы
        double pageHeight = format.getImageableHeight();

        // Если рисунок слишком широк или высок для данной
        // страницы - уменьшаем его.
        if (size.width > pageWidth) {
            // На какую величину уменьшить
            double factor = pageWidth/size.width;
            // Настройка координатной системы
            g2.scale(factor, factor);
            // Настройка размера бумаги
            pageWidth /= factor;
            pageHeight /= factor;
        }
        if (size.height > pageHeight) {   // То же самое выполняем для высоты
            double factor = pageHeight/size.height;
            g2.scale(factor, factor);
            pageWidth /= factor;
```

```
                pageHeight /= factor;
            }

            // Сейчас мы знаем, что рисунок поместится на странице.
            // При необходимости центрируем его
            g2.translate((pageWidth-size.width)/2,
                         (pageHeight-size.height)/2);

            // Рисуем линию по внешним сторонам области вывода
            g2.drawRect(-1, -1, size.width+2, size.height+2);

            // Устанавливаем область отсечки, так чтобы рисунок
            // не вышел за ее границы
            g2.setClip(0, 0, size.width, size.height);

            // Наконец, печатаем компонент с помощью вызова
            // метода paintComponent(). Или вызываем paint()
            // для отрисовки компонента, фона, рамки и дочерних
            // элементов, включая кнопку «Print»
            this.paintComponent(g);
            // Говорим объекту PrinterJob, что номер страницы верен
            return Printable.PAGE_EXISTS;
        }

        /** Этот метод вызывается, когда пользователь начинает рисовать линию */
        public void processMouseEvent(MouseEvent e) {
            if (e.getID() == MouseEvent.MOUSE_PRESSED) {
                // Начинаем новую линию
                scribble.moveTo(e.getX(), e.getY());
            }
            else super.processMouseEvent(e);
        }

        /** Этот метод вызывается, когда пользователь перемещает мышь:
            рисует линию */
        public void processMouseMotionEvent(MouseEvent e) {
            if (e.getID() == MouseEvent.MOUSE_DRAGGED) {
                // Добавляем линию к фрагменту
                scribble.lineTo(e.getX(), e.getY());
                // Перерисовываем весь фрагмент. Медленно очищаем.
                repaint();
            }
            else super.processMouseMotionEvent(e);
        }

        /** Метод main(). Создаем объект ScribblePrinter2 и выходим */
        public static void main(String[] args) {
            JFrame frame = new JFrame("ScribblePrinter2");
            ScribblePrinter2 s = new ScribblePrinter2();
            frame.getContentPane().add(s, BorderLayout.CENTER);
            frame.setSize(400, 400);
            frame.setVisible(true);
        }
    }
```

Печать многостраничных текстовых документов

В двух примерах, которые мы видели, графические компоненты распечатывались на одной странице. Печать многостраничных документов гораздо интереснее, но зато требует и определенных ухищрений, поскольку мы должны решить, где помещать разрывы страниц. В примере 12.3 показано, как это может быть сделано. Данный класс `HardcopyWriter` является производным потоком `java.io.Writer`, который использует интерфейс печати Java 1.1 для печати пересылаемых через него символов, вставляя при необходимости разрывы строк и страниц.

Класс `HardcopyWriter` включает две демонстрационные программы, реализованные в виде внутренних классов. Первая, `PrintFile`, читает указанный текстовый файл и распечатывает его, посылая его содержимое в поток `HardcopyWriter`. Вторая, `Demo`, распечатывает демонстрационную страницу, показывающую возможности этого класса по управлению шрифтами и табуляцией, как показано на рис. 12.1.

Рис. 12.1. *Демонстрационная страница, напечатанная программой HardcopyWriter*

Пример 12.3 довольно велик, но его стоит изучить. Кроме программного интерфейса печати Java 1.1 он демонстрирует прием разбиения на страницы текстовых документов. Он также является полезным примером использования производных потоков `Writer`.

Пример 12.3. HardcopyWriter.java

```
package com.davidflanagan.examples.print;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.text.*;
import java.util.*;

/**
 * Символьный выходной поток, посылающий вывод на принтер.
 **/
public class HardcopyWriter extends Writer {
    // Это переменные экземпляра класса
    protected PrintJob job;          // Используемый объект PrintJob
    protected Graphics page;         // Объект Graphics для текущей страницы
    protected String jobname;        // Имя задания печати
    protected int fontsize;          // Размер шрифта в пунктах
    protected String time;           // Текущее время (выводится в заголовке)
    protected Dimension pagesize;    // Размер страниц (в точках)
    protected int pagedpi;           // Разрешение страницы в точках на дюйм
    protected Font font, headerfont;      // Основной шрифт и шрифт заголовка
    protected FontMetrics metrics;         // Метрики для основного шрифта
    protected FontMetrics headermetrics;   // Метрики для шрифта заголовка
    protected int x0, y0;            // Верхний левый угол внутреннего поля
    protected int width, height;     // Размер (в точках) внутреннего поля
    protected int headery;           // Базовая линия заголовка страницы
    protected int charwidth;         // Ширина каждого символа
    protected int lineheight;        // Высота каждой строки
    protected int lineascent;        // Смещение базовой линии шрифта
    protected int chars_per_line;    // Количество знаков в строке
    protected int lines_per_page;    // Количество строк на странице
    protected int charnum = 0, linenum = 0; // Позиция текущих столбца и строки
    protected int pagenum = 0;       // Номер текущей страницы

    // Поле для хранения состояния в промежутках между вызовами
    // метода write()
    private boolean last_char_was_return = false;

    // Статическая переменная для хранения пользовательских
    // настроек между заданиями печати
    protected static Properties printprops = new Properties();

    /**
     * Конструктор для данного класса имеет много аргументов:
     * аргумент frame требуется для выполнения любой печати в Java;
     * jobname выводится вверху слева каждой печатаемой страницы;
     * размер шрифта задается в пунктах, как и у экранных шрифтов;
```

```
     * поля указываются в дюймах (или долях дюйма).
     **/
    public HardcopyWriter(Frame frame, String jobname,
        int fontsize, double leftmargin, double rightmargin,
        double topmargin, double bottommargin)
        throws HardcopyWriter.PrintCanceledException
    {
        // Получаем объект PrintJob, с помощью которого мы будем выполнять
        // всю печать. Вызов синхронизируется с помощью статического
        // объекта printprops. Это означает, что в каждый момент может быть
        // выведен только один диалог печати. Если пользователь нажал
        // кнопку «Cancel» в диалоге печати, генерируется исключение.
        Toolkit toolkit = frame.getToolkit();  // Получаем Toolkit
                                               // из объекта Frame
        synchronized(printprops) {
            job = toolkit.getPrintJob(frame, jobname, printprops);
        }
        if (job == null)
            throw new PrintCanceledException("Пользователь отменил запрос
                                             на печать");

        pagesize = job.getPageDimension();     // Запрашиваем размер страницы
        pagedpi = job.getPageResolution();     // Запрашиваем разрешение
                                               // страницы

        // Обработка ошибки:
        // В Windows методы getPageDimension() и getPageResolution
        // не работают, поэтому мы должны смоделировать их.
       if (System.getProperty("os.name").regionMatches(true,0,"windows",0,7)){
            // Используем разрешение экрана, которое PrintJob
            // пытается эмулировать
            pagedpi = toolkit.getScreenResolution();
            // Предполагаем, что размер бумаги равен 8.5 x 11 дюймов.
            // Пользователи, использующие формат A4, должны это изменить.
            pagesize = new Dimension((int)(8.5 * pagedpi), 11*pagedpi);
            // Также мы должны вычислить размер шрифта.
            // Он указывается в пунктах, (1 пункт = 1/72 дюйма),
            // но Windows измеряет его пикселами.
            fontsize = fontsize * pagedpi / 72;
        }

        // Вычисляем координаты верхнего левого угла страницы.
        // Т. е. координаты (leftmargin, topmargin). Также вычисляем ширину
        // и высоту области между полями.
        x0 = (int)(leftmargin * pagedpi);
        y0 = (int)(topmargin * pagedpi);
       width = pagesize.width - (int)((leftmargin + rightmargin) * pagedpi);
       height = pagesize.height - (int)((topmargin + bottommargin) * pagedpi);

        // Получаем основной шрифт и его размер
        font = new Font("Monospaced", Font.PLAIN, fontsize);
        metrics = frame.getFontMetrics(font);
        lineheight = metrics.getHeight();
```

```
        lineascent = metrics.getAscent();
        charwidth = metrics.charWidth('0');  // Предполагаем, что шрифт -
                                             // моноширинный

        // Сейчас вычисляем количество столбцов и строк,
        // которые уместятся между полями.
        chars_per_line = width / charwidth;
        lines_per_page = height / lineheight;

        // Получаем информацию о шрифте заголовка
        // и вычисляем базовую линию заголовка: 1/8 дюйма над верхним полем
        headerfont = new Font("SansSerif", Font.ITALIC, fontsize);
        headermetrics = frame.getFontMetrics(headerfont);
        headery = y0 - (int)(0.125 * pagedpi) -
            headermetrics.getHeight() + headermetrics.getAscent();

        // Формируем строку, содержащую дату и время для отображения
        // в заголовке страницы.
        DateFormat df = DateFormat.getDateTimeInstance(DateFormat.LONG,
                                                       DateFormat.SHORT);
        df.setTimeZone(TimeZone.getDefault());
        time = df.format(new Date());

        this.jobname = jobname;   // Сохраняем имя
        this.fontsize = fontsize; // Сохраняем размер шрифта
    }

    /**
     * Это метод  write() потока. Его реализуют все классы, производные от
     * Writer. Все остальные версии метода write() являются вариантами этого.
     **/
    public void write(char[] buffer, int index, int len) {
        synchronized(this.lock) {  // Для обеспечения безопасности потока
            // Цикл по всем переданным нам символам
            for(int i = index; i < index + len; i++) {
                // Если мы еще не начали страницу (или если это новая
                // страница), делаем это сейчас.
                if (page == null) newpage();

               // Если символ является символом конца строки, начинаем новую
                // строку, если это не \n, непосредственно следующий за \r.
                if (buffer[i] == '\n') {
                    if (!last_char_was_return) newline();
                    continue;
                }
                if (buffer[i] == '\r') {
                    newline();
                    last_char_was_return = true;
                    continue;
                }
                else last_char_was_return = false;

              // Если это какой-либо другой непечатный символ - пропускаем его.
                if (Character.isWhitespace(buffer[i]) &&
```

```
                !Character.isSpaceChar(buffer[i]) && (buffer[i] != '\t'))
                continue;

            // Если в строку больше не поместится ни одного символа,
            // начинаем новую строку.
            if (charnum >= chars_per_line) {
                newline();
                // При необходимости также начинаем новую страницу.
                if (page == null) newpage();
            }

 // Сейчас печатаем символ:
 // Если это пробел - пропускаем один пробел, не выполняя вывод.
 // Если это символ табуляции - пропускаем необходимое число пробелов.
 // В других случаях - печатаем символ. Неэффективно печатать
 // только по одному символу за раз, но из-за того, что наши
 // значения FontMetrics не совпадают в точности с используемыми
 // принтером, нам требуется позиционировать каждый символ отдельно.
            if (Character.isSpaceChar(buffer[i])) charnum++;
            else if (buffer[i] == '\t') charnum += 8 - (charnum % 8);
            else {
                page.drawChars(buffer, i, 1,
                   x0 + charnum*charwidth,
                   y0 + (linenum*lineheight) + lineascent);
                charnum++;
            }
        }
    }
}

/**
 * Это метод flush(), который должны реализовать все классы,
 * производные от Writer. Нет способа сбросить объект PrintJob
 * без преждевременной печати страницы, поэтому мы ничего не делаем
 **/
public void flush() { /* ничего не делаем */ }

/**
 * Это метод close(), который должны реализовать
 * все классы, производные от Writer. Печатает текущую
 * страницу (если такая есть) и завершает PrintJob.
 */
public void close() {
    synchronized(this.lock) {
       if (page != null) page.dispose();  // Посылаем страницу на принтер
        job.end();                        // Завершаем задачу
    }
}

/**
 * Устанавливаем начертание шрифта: аргумент должен быть
 * одной из констант, соответствующих стилям шрифта, определенных
 * в классе java.awt.Font. Весь последующий вывод будет выполняться
```

```
     * с использованием этого начертания. Этот метод относится ко всем
     * начертаниям моноширинного шрифта, имеющим такие же метрики
     **/
    public void setFontStyle(int style) {
        synchronized (this.lock) {
            // Пытаемся установить новый шрифт, но восстанавливаем текущий,
            // если нам это не удалось.
            Font current = font;
            try { font = new Font("Monospaced", style, fontsize); }
            catch (Exception e) { font = current; }
            // Если существует текущая страница, устанавливаем новый шрифт.
            // Иначе это сделает метод newpage().
            if (page != null) page.setFont(font);
        }
    }

    /** Завершаем текущую страницу. Последующий вывод.
     будет выполняться на новой странице */
    public void pageBreak() { synchronized(this.lock) { newpage(); } }

    /** Возвращаем количество символов, которые помещаются в строке страницы */
    public int getCharactersPerLine() { return this.chars_per_line; }

    /** Возвращаем количество строк, которые помещаются на странице */
    public int getLinesPerPage() { return this.lines_per_page; }

    /** Этот внутренний метод начинает новую строку */
    protected void newline() {
        charnum = 0;  // устанавливаем количество символов в 0
        linenum++;    // Увеличиваем номер строки
        if (linenum >= lines_per_page) {  // Если мы достигли конца страницы,
            page.dispose();               // посылаем страницу на принтер,
            page = null;                  // но еще не начинаем новую страницу.
        }
    }

    /** Этот внутренний метод начинает новую страницу и печатает заголовок. */
    protected void newpage() {
        page = job.getGraphics();   // Начинаем новую страницу
        linenum = 0; charnum = 0;   // Сбрасываем номера строки и символа
        pagenum++;                  // Увеличиваем номер страницы
        page.setFont(headerfont);   // Устанавливаем шрифт заголовка.
        page.drawString(jobname, x0, headery); // Печатаем выровненное
                                               // по левому краю имя задания
        String s = "- " + pagenum + " -";      // Печатаем центрированный
                                               // номер страницы.
        int w = headermetrics.stringWidth(s);
        page.drawString(s, x0 + (this.width - w)/2, headery);
        w = headermetrics.stringWidth(time);   // Печатаем выровненную
                                               // по правому краю дату
        page.drawString(time, x0 + width - w, headery);

        // Рисуем линию под заголовком
```

```
    int y = headery + headermetrics.getDescent() + 1;
    page.drawLine(x0, y, x0+width, y);

    // Устанавливаем основной моноширинный шрифт до конца страницы.
    page.setFont(font);
}

/**
 * Это класс исключения, которое генерируется
 * конструктором HardcopyWriter при нажатии пользователем
 * кнопки «Cancel» в диалоговом окне печати
 **/
public static class PrintCanceledException extends Exception {
    public PrintCanceledException(String msg) { super(msg); }
}

/**
 * Программа, печатающая указанный файл с помощью HardcopyWriter
 **/
public static class PrintFile {
    public static void main(String[] args) {
        try {
            if (args.length != 1)
                throw new IllegalArgumentException("Неверное число
                                                    аргументов");
            FileReader in = new FileReader(args[0]);
            HardcopyWriter out = null;
            Frame f = new Frame("PrintFile: " + args[0]);
            f.setSize(200, 50);
            f.show();
            try {
                out = new HardcopyWriter(f, args[0], 10, .5, .5, .5, .5);
            }
            catch (HardcopyWriter.PrintCanceledException e) {
                System.exit(0);
            }
            f.setVisible(false);
            char[] buffer = new char[4096];
            int numchars;
            while((numchars = in.read(buffer)) != -1)
                out.write(buffer, 0, numchars);
            in.close();
            out.close();
        }
        catch (Exception e) {
            System.err.println(e);
            System.err.println("Формат: java " +
              "HardcopyWriter$PrintFile <имя файла>");
            System.exit(1);
        }
        System.exit(0);
    }
```

```
}

/**
 * Программа, которая печатает демонстрационную страницу HardcopyWriter
 **/
public static class Demo extends Frame implements ActionListener {
    /** Метод main() программы. Создает тестовое окно */
    public static void main(String[] args) {
        Frame f = new Demo();
        f.show();
    }
    // Кнопки, используемые в программе
    protected Button print, quit;

    /** Конструктор для окна тестовой программы. */
    public Demo() {
        super("HardcopyWriter Test");       // Вызываем конструктор окна
        Panel p = new Panel();              // Добавляем в окно панель
        this.add(p, "Center");              // Центрируем ее
        p.setFont(new Font("SansSerif",     // Устанавливаем шрифт
                    Font.BOLD, 18));        // по умолчанию
        print = new Button("Print Test Page"); // Создаем кнопку "Print"
        quit = new Button("Quit");          // Создаем кнопку "Quit"
        // Указываем, что мы будем обрабатывать
        // нажатия кнопок.
        print.addActionListener(this);
        quit.addActionListener(this);
        p.add(print);  // Добавляем кнопки в панель
        p.add(quit);
        this.pack();   // Устанавливаем размер окна
    }

    /** Обрабатываем нажатия кнопок */
    public void actionPerformed(ActionEvent e) {
        Object o = e.getSource();
        if (o == quit) System.exit(0);
        else if (o == print) printDemoPage();
    }

    /** Печатаем демонстрационную страницу */
    public void printDemoPage() {
        // Создаем объект HardcopyWriter, используя
        // шрифт размером 14 пунктов в 3/4 дюйма.
        HardcopyWriter hw;
        try { hw=new HardcopyWriter(this, "Демонстрационная страница" ,
                                                14,.75,.75,.75,.75);}
        catch (HardcopyWriter.PrintCanceledException e) { return; }

        //  Посылаем вывод через поток PrintWriter
        PrintWriter out = new PrintWriter(hw);

        // Выясняем размер страницы
       int rows = hw.getLinesPerPage(), cols = hw.getCharactersPerLine();
```

```
// Отмечаем верхний левый и верхний правый углы
out.print("+");                              // Верхний левый угол
for(int i=0;i<cols-2;i++) out.print(" "); // Пропускаем место
out.print("+");                              // Верхний правый угол

// Отображаем заголовок
hw.setFontStyle(Font.BOLD + Font.ITALIC);
out.println("\n\t\t Демонстрационная страница
                        Hardcopy Writer \n\n");

// Демонстрируем стили шрифтов
hw.setFontStyle(Font.BOLD);
out.println("Стили шрифтов:");
int[] styles = { Font.PLAIN, Font.BOLD,
                 Font.ITALIC, Font.ITALIC+Font.BOLD };
for(int i = 0; i < styles.length; i++) {
    hw.setFontStyle(styles[i]);
    out.println("ABCDEFGHIJKLMNOPQRSTUVWXYZ" +
                "abcdefghijklmnopqrstuvwxyz");
    out.println("1234567890!@#$%^&*()[]{}<>,.?:;+-=/\\``\"_~|");
}
hw.setFontStyle(Font.PLAIN);
out.println("\n");

// Демонстрируем позиции табуляции
hw.setFontStyle(Font.BOLD);
out.println("Позиции табуляции:");
hw.setFontStyle(Font.PLAIN);
out.println("         1         2         3         4         5");
out.println("012345678901234567890123456789012345678901234567890");
out.println("^\t^\t^\t^\t^\t^");
out.println("\n");

// Выводим информацию о разрешении и других характеристиках страницы
hw.setFontStyle(Font.BOLD);
out.println("Характеристики:");
hw.setFontStyle(Font.PLAIN);
out.println("\t Разрешение: " + hw.pagedpi + " точек на дюйм");
out.println("\t Ширина страницы (пикселы): " + hw.pagesize.width);
out.println("\t Высота страницы (пикселы): " + hw.pagesize.height);
out.println("\t Ширина внутренней области (пикселы): " + hw.width);
out.println("\t Высота внутренней области (пикселы): " + hw.height);
out.println("\t Символов в строке: " + cols);
out.println("\t Строк на странице: " + rows);

// Опускаемся до конца страницы
for(int i = 0; i < rows-30; i++) out.println();

// И отмечаем нижние левый и правый углы
out.print("+");      // Левый нижний
for(int i=0;i<cols-2;i++)
    out.print(" ");  // Пропуск пробелов
out.print("+");      // Правый нижний
```

```
                // Закрываем выходной поток, что приводит к печати страницы
                out.close();
            }
        }
    }
```

Печать Swing-документов

В примере 10.21 главы 10 были продемонстрированы возможности компонента javax.swing.text.JTextComponent по отображению сложных документов, включая HTML-документы. Однако одной из возможностей, отсутствующей в JTextComponent, является способность печати этих документов. В примере 10.21 заложена возможность печати, основанная на классе PrintableDocument, листинг которого приведен в примере 12.4.

PrintableDocument использует интерфейс печати Java 1.2. Помимо интерфейса Printable в примере реализован интерфейс Pageable. Pageable представляет многостраничный документ и определяет три метода: getNumberOfPages(), возвращающий количество страниц в документе; getPageFormat(), возвращающий размер и ориентацию каждой страницы документа; getPrintable(), который возвращает объект Printable, представляющий каждую страницу.

Помимо демонстрации использования интерфейсов Pageable и Printable в примере показана внутренняя работа пакета javax.swing.text. В этом пакете объекты Document составлены из вложенных объектов Element, отображаемых на экран (или принтер) с помощью параллельной иерархии объектов View. javax.swing.text относится к наиболее сложным областям библиотеки Swing, поэтому не переживайте, если вы поймете не все его детали.

К сожалению, класс PrintableDocument не является настолько мощным и надежным, как мне бы хотелось. В своих тестах я обнаружил, что он произвольно вставляет разрыв страницы в середину текстовой строки, помещая половину строки внизу одной страницы, а остаток – вверху следующей. Я не уверен, где ошибка – в классе PrintableDocument или в пакете javax.swing.text. Кроме этого, PrintableDocument недостаточно хорошо обрабатывает непрерывные длинные строки. Если документ содержит неразрывную строку длиной более запрошенной ширины страницы (например, HTML-тег <PRE>), класс Document делает ширину всего документа равной длине самой большой строки. Таким образом, происходит обрезание не только длинных строк, находящихся внутри тега <PRE>, – весь документ форматируется слишком широко, что приводит к обрезанию всех строк по правому полю печатаемой страницы.

Пример 12.4. PrintableDocument.java

```
package com.davidflanagan.examples.print;
import java.awt.*;
```

```
import java.awt.print.*;
import java.awt.geom.*;
import java.awt.font.*;
import javax.swing.*;
import javax.swing.text.*;
import java.util.*;

/**
 * Этот класс реализует интерфейсы Pageable и Printable
 * и позволяет распечатывать содержимое любого компонента
 * JTextComponent, используя интерфейс печати java.awt.print
 **/
public class PrintableDocument implements Pageable, Printable {
    View root;            // корневой объект View, подлежащий печати
    PageFormat format;      // Бумага и ориентация страницы
    double scalefactor;     // Коэффициент масштабирования перед печатью
    int numPages;           // Количество страниц в документе
    double printX, printY;  // Координаты верхнего левого угла области печати
    double printWidth;      // Ширина печатаемой области
    double printHeight;     // Высота печатаемой области
    Rectangle drawRect;   // Прямоугольник, в котором будет рисоваться документ

    // Степень варьирования нижнего поля для предотвращения висячих строк
    static final double MARGIN_ADJUST = .97;

    // Шрифт, используемый для печати номеров страниц
    static final Font headerFont = new Font("Serif", Font.PLAIN, 12);

    /**
     * Этот конструктор позволяет печатать содержимое любого
     * компонента JtextComponent, используя принятые по умолчанию
     * объект PageFormat и коэффициент масштабирования.
     * Коэффициент масштабирования по умолчанию - 0.75,
     * так как используемый по умолчанию шрифт - очень большой
     */
    public PrintableDocument(JTextComponent textComponent) {
        this(textComponent, new PageFormat(), .75);
    }

    /**
     * Этот конструктор позволяет печатать содержимое любого
     * компонента JTextComponent, используя указанный
     * PageFormat и любой коэффициент масштабирования.
     **/
    public PrintableDocument(JTextComponent textComponent,
             PageFormat format, double scalefactor)
    {
       // Запоминаем формат страницы и запрашиваем его для печатаемой области
        this.format = format;
        this.scalefactor = scalefactor;
        this.printX = format.getImageableX()/scalefactor;
        this.printY = format.getImageableY()/scalefactor;
        this.printWidth = format.getImageableWidth()/scalefactor;
```

```
        this.printHeight = format.getImageableHeight()/scalefactor;
        double paperWidth = format.getWidth()/scalefactor;

        // Получаем документ и его корневой объект Element
        // из текстового компонента
        Document document = textComponent.getDocument();
        Element rootElement = document.getDefaultRootElement();
        // Получаем из текстового компонента EditorKit его ViewFactory
       EditorKit editorKit =textComponent.getUI().getEditorKit(textComponent);
        ViewFactory viewFactory = editorKit.getViewFactory();

        // Используем ViewFactory для создания корневого объекта View
        // для документа. Он является объектом, который мы печатаем.
        root = viewFactory.create(rootElement);

        // Текстовая архитектура Swing требует, чтобы мы вызвали для нашего
        // корневого View метод setParent() перед началом его использования.
        // Для того чтобы сделать это, нам нужен объект View,
        // который может послужить родителем.

        // Мы используем пользовательскую реализацию, определенную ниже.
        root.setParent(new ParentView(root, viewFactory, textComponent));

        // Сообщаем представлению View ширину страницы, оно должно
        // отформатировать себя так, чтобы уместиться в эту ширину.
        // Высота не имеет сейчас значения.
        root.setSize((float)printWidth, (float)printHeight);

        // Сейчас, когда представление отформатировало себя
        // в соответствии с заданной шириной, спрашиваем его о высоте
         double documentHeight = root.getPreferredSpan(View.Y_AXIS);

        // Формируем прямоугольник, сообщающий представлению, где себя
        // рисовать. Мы будем использовать его в других методах
        // данного класса.
       drawRect = new Rectangle(0, 0, (int)printWidth, (int)documentHeight);

        // Сейчас, если документ длиннее одной страницы,
        // нам нужно определить, где нужны разрывы страниц
        if (documentHeight > printHeight) paginate(root, drawRect);

        // После того как мы разбили его на страницы,
        // выясняем, сколько их получилось.
        numPages = pageLengths.size() + 1;
    }

    // Это начальное смещение страницы, над которой
    // мы в настоящее время работаем.
    double pageStart = 0;

    /**
     * Этот метод проходит в цикле дочерние элементы указанного
     * представления, выполняя при необходимости рекурсивные вызовы,
     * и вставляет разрывы страниц там, где они нужны.
     * Он выполняет простейшие операции для предотвращения висячих строк.
     **/
```

```
protected void paginate(View v, Rectangle2D allocation) {
    // Выясняем высоту представления и даем ему команду
    // распределить это пространство между его дочерними элементами
    double myheight = v.getPreferredSpan(View.Y_AXIS);
    v.setSize((float)printWidth, (float)myheight);

    // Сейчас выполняем цикл по всем дочерним элементам
    int numkids = v.getViewCount();
    for(int i = 0; i < numkids; i++) {
        View kid = v.getView(i);  // Это элемент, с которым мы работаем
        // Определим его размер и положение
        Shape kidshape = v.getChildAllocation(i, allocation);
        if (kidshape == null) continue;
        Rectangle2D kidbox = kidshape.getBounds2D();

        // Это координата Y нижнего края элемента
        double kidpos = kidbox.getY() + kidbox.getHeight() - pageStart;

        // Если это первый элемент в группе, нам нужно
        // убедиться, что он не останется один внизу страницы.
        // То есть мы хотим предотвратить нижние висячие строки
        if ((numkids > 1) && (i == 0)) {
            // Если мы не находимся рядом с концом страницы,
            // то просто переходим к следующему элементу.
            if (kidpos < printY + printHeight*MARGIN_ADJUST) continue;

            // Иначе элемент находится рядом с низом страницы,
            // поэтому разрываем страницу перед этим элементом
            // и помещаем его на следующую страницу.
            breakPage(kidbox.getY());
            continue;
        }

        // Если это последний элемент группы, мы не хотим, чтобы он
        // появился один наверху новой страницы, поэтому при необходимости
        // разрешаем ему отодвинуть границу нижнего поля.
        // Это позволит нам избежать верхних висячих строк
        if ((numkids > 1) && (i == numkids-1)) {
            // Если он размещается нормально, просто
            // переходим к следующему
            if (kidpos < printY + printHeight) continue;

            // Иначе, если он требует слишком много места,
            // разрываем страницу по концу группы.
            if (kidpos < printY + printHeight/MARGIN_ADJUST) {
                breakPage(allocation.getY() + allocation.getHeight());
                continue;
            }
        }

        // Если элемент не является ни первым, ни последним элементом
        // группы, мы используем точно заданную нижнюю границу.
        // Если элемент умещается на странице – переходим к следующему.
        if (kidpos < printY+printHeight) continue;
```

```
            // Если мы попали сюда, значит, элемент не умещается на странице.
            // Если у него нет дочерних элементов, вставляем перед ним
            // разрыв страницы и продолжаем.
            if (kid.getViewCount() == 0) {
                breakPage(kidbox.getY());
                continue;
            }

            // Если мы попали сюда, значит, элемент не уместился
            // на странице. У него есть дочерние элементы, поэтому делаем
            // рекурсивный вызов, чтобы выяснить, поместятся ли на этой
            // странице какие-либо из его дочерних элементов.
            paginate(kid, kidbox);
        }
    }

    // Для документа, состоящего из n страниц, этот список
    // будет хранить длины страниц с номерами от 0 до n-2.
    // Предполагается, что последняя страница имеет полную длину.
    ArrayList pageLengths = new ArrayList();

    // Для документа, состоящего из n страниц, этот список
    // хранит начальное смещение с 1 по n-1.
    // Смещение страницы 0 всегда равно 0.
    ArrayList pageOffsets = new ArrayList();

    /**
     * Разрываем страницу по указанной координате Y.
     * Запоминаем необходимую информацию в списках
     * pageLength и pageOffsets.
     **/
    void breakPage(double y) {
        double pageLength = y-pageStart-printY;
        pageStart = y-printY;
        pageLengths.add(new Double(pageLength));
        pageOffsets.add(new Double(pageStart));
    }

    /** Возвращаем количество страниц. Это метод интерфейса Pageable */
    public int getNumberOfPages() { return numPages; }

    /**
     * Возвращаем для указанной страницы объект PageFormat.
     * Это метод Pageable. Эта реализация использует
     * вычисленную длину страницы в возвращенном объекте
     * PageFormat. PrinterJob будет использовать ее в качестве
     * области отсечки, которая предотвратит вывод в верхнее
     * и нижнее поля посторонних фрагментов документа.
     **/
    public PageFormat getPageFormat(int pagenum) {
        // На последней странице просто возвращаем формат
        // страницы, заданный пользователем.
        if (pagenum == numPages-1) return format;
```

```
        // Иначе возвращаем соответствующий PageFormat
        // с учетом высоты данной страницы.
        double pageLength = ((Double)pageLengths.get(pagenum)).doubleValue();
        PageFormat f = (PageFormat) format.clone();
        Paper p = f.getPaper();
        if (f.getOrientation() == PageFormat.PORTRAIT)
            p.setImageableArea(printX*scalefactor,
               printY*scalefactor, printWidth*scalefactor,
               pageLength*scalefactor);
        else
            p.setImageableArea(printY*scalefactor,
              printX*scalefactor, pageLength*scalefactor,
              printWidth*scalefactor);
        f.setPaper(p);
        return f;
    }

    /**
     * Этот метод Pageable возвращает для указанной страницы
     * объект Printable. Так как данный класс реализует
     * и Pageable, и Printable, он просто возвращает this.
     **/
    public Printable getPrintable(int pagenum) { return this; }

    /**
     * Это основной метод интерфейса Printable, печатающий указанную страницу
     **/
    public int print(Graphics g, PageFormat format, int pageIndex) {
        // При попытке печати после конца документа возвращает код ошибки

        if (pageIndex >= numPages) return NO_SUCH_PAGE;

        // Выполняем приведение объекта Graphics, так чтобы
        // мы могли использовать операции Java2D
        Graphics2D g2 = (Graphics2D)g;

        // Выполняем преобразование для подгонки верхнего и левого полей
        g2.translate(format.getImageableX(), format.getImageableY());

        // Масштабируем страницу, используя указанный
        // коэффициент масштабирования
        g2.scale(scalefactor, scalefactor);

        // Отображаем номер страницы по центру области верхнего поля.
        // Устанавливаем новую область отсечки, чтобы мы могли рисовать
        // внутри верхнего поля. Но запоминаем первоначальную
        // область отсечки, чтобы мы могли восстановить ее.
        if (pageIndex > 0) {
            Shape originalClip = g.getClip();
            g.setClip(new Rectangle(0, (int)-printY,
                      (int)printWidth, (int)printY));
            // Формируем отображаемый заголовок, измеряем и выводим его
            String numString = "- " + (pageIndex+1) + " -";
            // Получаем размеры строки и шрифта
```

```
            FontRenderContext frc = g2.getFontRenderContext();
            Rectangle2D numBounds = headerFont.getStringBounds(numString, frc);
            LineMetrics metrics = headerFont.getLineMetrics(numString, frc);
            g.setFont(headerFont);    // Устанавливаем шрифт
            g.setColor(Color.black);  // Печатаем черными чернилами
            g.drawString(numString,   // Отображаем строку
               (int)((printWidth-numBounds.getWidth())/2),
              (int)(-(printY-numBounds.getHeight())/2 + metrics.getAscent()));
            g.setClip(originalClip);  // Восстанавливаем область отсечки
        }

        // Получаем начальную позицию и длину страницы внутри документа
        double pageStart = 0.0, pageLength = printHeight;
        if (pageIndex > 0)
            pageStart = ((Double)pageOffsets.get(pageIndex-1)).doubleValue();
        if (pageIndex < numPages-1)
            pageLength = ((Double)pageLengths.get(pageIndex)).doubleValue();

        // Смещаем соответствующую часть документа так,
        // чтобы выровнять ее по верхнему левому углу страницы
        g2.translate(0.0, -pageStart);

        // Сейчас выводим весь документ. Благодаря применению области
        // отсечки на данном листе бумаги в действительности будет
        // выведен только нужный фрагмент документа
        root.paint(g, drawRect);

        // В конце возвращаем код успешного выполнения операции
        return PAGE_EXISTS;
    }

    /**
     * Этот внутренний класс является конкретной реализацией
     * объекта View с парой ключевых методов реализации. Экземпляр данного
     * класса используется в качестве родительского элемента корневого
     * объекта View, который мы хотим напечатать
     **/
    static class ParentView extends View {
        ViewFactory viewFactory; // ViewFactory для иерархии представлений
        Container container;     // контейнер для иерархии представлений

        public ParentView(View v, ViewFactory viewFactory, Container container)
        {
            super(v.getElement());
            this.viewFactory = viewFactory;
            this.container = container;
        }

        // Эти методы возвращают ключевые фрагменты информации,
        // необходимые для иерархии представлений
        public ViewFactory getViewFactory() { return viewFactory; }
        public Container getContainer() { return container; }

        // Эти методы объекта View являются абстрактными, поэтому здесь
```

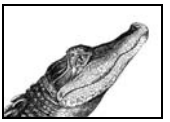
```
            // мы должны предоставить для них фиктивную реализацию,
            // даже несмотря на то, что они никогда не будут использованы.
            public void paint(Graphics g, Shape allocation) {}
            public float getPreferredSpan(int axis) { return 0.0f; }
            public int viewToModel(float x,float y,Shape a,Position.Bias[] bias)
{
                return 0;
            }
            public Shape modelToView(int pos, Shape a, Position.Bias b)
                throws BadLocationException {
                return a;
            }
        }
}
```

Упражнения

12-1. В Java 1.3 `java.awt.JobAttributes` и `java.awt.PageAttributes` являются новыми классами, расширяющими интерфейс печати Java 1.1. Изучите документацию по этим классам, а затем измените класс `HardcopyWriter` так, чтобы можно было передавать объекты `PageAttributes` и `JobAttributes` в конструктор `HardcopyWriter`. Протестируйте модифицированный класс, написав программу, печатающую в режиме альбомной ориентации страницы и использующую двустороннюю печать (если ваш принтер поддерживает такую возможность).

12-2. С помощью интерфейса печати Java 1.2 метод `print()` объекта `Printable` может вызываться несколько раз для каждой страницы, чтобы обеспечить печать изображений и рисунков с высоким разрешением. По этой причине в Java 1.2 невозможно напечатать символьный поток, как это сделано в классе `HardcopyWriter` Java с помощью интерфейса Java 1.1. Тем не менее с помощью интерфейса Java 1.2 все же можно печатать многостраничные текстовые документы. Напишите программу, читающую текстовый файл и печатающую его содержимое с использованием программного интерфейса Java 1.2.

12-3. Модифицируйте вашу программу из предыдущего примера так, чтобы она поддерживала печать расположенных рядом страниц, уменьшенных в два раза, в альбомной ориентации. Для выполнения этого вам потребуется использовать средства Java по двумерному масштабированию и вращению.



Глава 13

Передача данных

В Java версии 1.1 пакет java.awt.datatransfer предоставляет возможность передачи данных между приложениями и поддерживает метод обмена данными типа «вырезание и вставка» (cut-and-paste). Java 2.0 расширяет возможности обмена данными графических приложений, добавляя пакет java.awt.dnd, поддерживающий метод передачи данных типа «перетаскивание» (drag-and-drop). В этой главе показано, как можно добавить в ваши AWT- и Swing-приложения поддержку методов вырезания–вставки и перетаскивания. В ней также показано, как использовать класс DataFlavor и интерфейс Tranferable для реализации передачи между приложениями произвольных типов данных.

Архитектура передачи данных

Перед тем как начать обсуждение примеров передачи данных, важно понять архитектуру передачи данных, используемую пакетом java.awt.datatransfer (и java.awt.dnd). Класс java.awt.datatransfer.DataFlavor, пожалуй, является центральным; он представляет тип данных, подлежащих передаче. Каждый формат данных (data flavor) содержит удобочитаемое имя, объект Class, указывающий Java-тип передаваемых данных, и тип MIME, определяющий кодировку, используемую при передаче данных. В классе DataFlavor предопределена пара наиболее часто используемых форматов для передачи строк и списков объектов File. Кроме этого, в нем определено несколько типов MIME, используемых этими форматами. Например, DataFlavor.stringFlavor может передавать объекты Java String в виде текста в формате Unicode. Он содержит класс представления java.lang.String и тип MIME:

```
application/x-java-serialized-object; class=java.lang.String
```

Интерфейс java.awt.datatransfer.Transferable является еще одной важной частью механизма передачи данных. Этот интерфейс задает три метода, которые должны быть реализованы каждым объектом, желающим сделать свои данные доступными для передачи: getTransferDataFlavor(), возвращающий массив всех типов DataFlavor, которые он может использовать для передачи своих данных; isDataFlavorSupported(), который проверяет, поддерживает ли объект Transferable данный формат, и наиболее важный метод getTransferData(), возвращающий данные в формате, соответствующем запрошенному DataFlavor.

Архитектура передачи данных основывается на механизме сериализации объектов как на одном из средств передачи данных между приложениями. Это означает, что архитектура передачи данных – достаточно общая и гибкая. Она была разработана для того, чтобы обеспечить возможность обмена произвольными данными между независимыми виртуальными машинами Java и даже между приложениями Java и платформенно-зависимыми приложениями. К сожалению, очевидно, что архитектура передачи данных не может быть реализована полностью, поэтому передача данных между машинами JVM и между JVM и «родным» приложением будет работать только в случае, если вы используете специальные предопределенные форматы – DataFlavor.stringFlavor и DataFlavor.javaFileListFlavor. Хотя вы и можете определять другие объекты DataFlavor, представляющие другие сериализуемые классы Java, эти пользовательские пакеты будут работать только внутри одной JVM.

Простое копирование и вставка

В то время как DataFlavor и Transferable предоставляют базовую инфраструктуру передачи данных, класс java.awt.datatransfer.Clipboard и интерфейс java.awt.datatransfer.ClipboardOwner поддерживают обмен данными в стиле «вырезание и вставка». Типичный сценарий вырезания и вставки работает примерно так:

- Когда пользователь дает команду скопировать или вырезать что-либо, активное приложение, прежде всего, получает системный объект Clipboard, используя вызов метода getSystemClipboard() объекта java.awt.Toolkit. Затем приложение создает объект Transferable, представляющий передаваемые данные. И наконец, оно передает этот объект буферу обмена с помощью вызова метода setContents() этого буфера. Активное приложение, кроме этого, должно передать в setContents() объект, реализующий интерфейс ClipboardOwner. Сделав это, приложение становится владельцем буфера обмена и должно поддерживать свой объект Transferable до тех пор, пока не перестанет быть владельцем.
- Когда пользователь дает команду вставки, целевое приложение, прежде всего, получает системный объект Clipboard таким же образом, как это делало активное приложение. Затем оно вызывает ме-

тод `getContents()` системного буфера обмена для получения хранящегося там объекта `Transferable`. Теперь оно может использовать методы, определяемые интерфейсом `Transferable` для выбора объекта `DataFlavor`, используемого для обмена данными, и фактически выполнить прием данных.

- Когда пользователь копирует или вырезает другой фрагмент данных, инициируется новая передача данных, и новое активное приложение (оно может быть тем же) становится новым владельцем буфера обмена. Предыдущий владелец уведомляется о том, что он больше не является владельцем буфера обмена, когда система вызывает метод `lostOwnership()` объекта `ClipboardOwner`, указанного при начальном вызове метода `setContents()`.

Обратите внимание на то, что ненадежным апплетам не разрешается работать с системным буфером обмена из-за того, что там могут находиться важные данные других приложений. Это значит, что апплеты не могут принимать участие в обмене данными типа «вырезание и вставка» между приложениями. Вместо этого апплеты должны создавать свой закрытый буфер обмена для использования обмена данными между апплетами.

Пример 13.1 представляет собой простую программу на базе AWT, демонстрирующую поддержку возможностей механизма вырезания и вставки в приложении. Она полагается на предопределенный объект `DataFlavor.stringFlavor` и класс `StringSelection`, реализующий интерфейс `Transferable` для строковых данных. Кроме этого, в программе использован объект `DataFlavor.javaFileListFlavor` для разрешения вставки имен файлов. Обратите внимание, что программа выполняет операции копирования, а не вырезания. Для того чтобы сделать вырезание, просто удалите текст из источника после его копирования в буфер обмена. Довольно интересными являются методы `copy()` и `paste()`, расположенные ближе к концу примера. Заметьте также, что пример реализует интерфейс `ClipboardOwner`, поэтому он получает уведомление, когда его данные больше не находятся в буфере обмена. Лучший способ протестировать данную программу – запустить две ее независимые копии и передать данные между ними. Вы также можете попытаться передать данные между экземпляром данной программы и родным для вашей платформы приложением.

Пример 13.1. SimpleCutAndPaste.java

```
package com.davidflanagan.examples.datatransfer;
import java.awt.*;
import java.awt.event.*;
import java.awt.datatransfer.*;

/**
 * Эта программа демонстрирует, как добавить в приложение
 * возможность "копирования и вставки"
```

```
 **/
public class SimpleCutAndPaste extends Frame implements ClipboardOwner
{
  /** Метод main() создает окно и отображает его. */
  public static void main(String[] args) {
    Frame f = new SimpleCutAndPaste();
    f.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) { System.exit(0); }
      });
    f.pack();
    f.setVisible(true);
  }

  /** Текстовое поле, хранящее вырезаемый или вставляемый текст */
  TextField field;

  /**
   * Конструктор создает очень простой тестовый графический
   * интерфейс и регистрирует его в качестве слушателя событий от кнопок
   **/
  public SimpleCutAndPaste() {
    super("SimpleCutAndPaste"); // Заголовок окна
    this.setFont(new Font("SansSerif", Font.PLAIN, 18)); // Используем
                                                         // крупный шрифт
    // Создаем кнопку "Cut"
    Button copy = new Button("Copy");
    copy.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) { copy(); }
      });
    this.add(copy, "West");

    // Создаем кнопку "Paste"
    Button paste = new Button("Paste");
    paste.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) { paste(); }
      });
    this.add(paste, "East");

    // Создаем текстовое поле, с которым они обе работают
    field = new TextField();
    this.add(field, "North");
  }

  /**
   * Этот метод принимает текущее содержимое текстового
   * поля, создает объект StringSelection для представления
   * этой строки и помещает StringSelection в буфер обмена
   **/
  public void copy() {
    // Получаем текущее отображаемое значение
    String s = field.getText();

    // Создаем объект StringSelection для представления этого текста.
    // StringSelection является предопределенным классом, реализующим для нас
```

```
        // интерфейсы Transferable и ClipboardOwner.
        StringSelection ss = new StringSelection(s);

        // Сейчас делаем объект StringSelection содержимым буфера обмена.
        // А также указываем, что мы являемся владельцем буфера обмена.
        // Выделяем текст, указывая на то, что он находится в буфере обмена.
        field.selectAll();
    }

    /**
     * Получаем содержимое буфера обмена и, если мы распознаем
     * его тип, отображаем его содержимое. Этот метод распознает
     * строки и списки файлов.
     **/
    public void paste() {
        // Получаем буфер обмена
        Clipboard c = this.getToolkit().getSystemClipboard();

        // Получаем содержимое буфера обмена в виде объекта Transferable
        Transferable t = c.getContents(this);

        // Выясняем, какой тип данных находится в буфере обмена
        try {
            if (t.isDataFlavorSupported(DataFlavor.stringFlavor)) {
                // Если это строка - получаем и отображаем строку
                String s = (String) t.getTransferData(DataFlavor.stringFlavor);
                field.setText(s);

            }
            else if (t.isDataFlavorSupported(DataFlavor.javaFileListFlavor)) {
                // Если это список объектов File - получаем этот список
                // и отображаем имя первого файла списка
                java.util.List files = (java.util.List)
                    t.getTransferData(DataFlavor.javaFileListFlavor);
                java.io.File file = (java.io.File)files.get(0);
                field.setText(file.getName());
            }
        }
        // Если при передаче что-либо происходит не так,
        // просто выдаем звуковой сигнал и ничего не делаем.
        catch (Exception e) { this.getToolkit().beep(); }
    }

    /**
     * Этот метод реализует интерфейс ClipboardOwner. Он вызывается,
     * когда в буфер обмена помещается что-либо еще.
     **/
    public void lostOwnership(Clipboard c, Transferable t) {
        // Снимаем выделение с текстового поля, так как мы больше не "владеем"
        // буфером обмена, и данный текст больше не доступен для вставки.
        field.select(0,0);
    }
}
```

Тип данных Transferable

Класс `java.awt.datatransfer.StringSelection` упрощает обмен строковыми значениями между приложениями. Однако для передачи других типов данных вы должны создать собственную реализацию интерфейса `Transferable`. В примере 13.2 показан класс `Scribble` – тип данных, представляющий набор отрезков линий. Он реализует интерфейс `Shape`, поэтому может быть отображен с помощью программного интерфейса Java 2D (см. главу 11 «Графика»). Но наиболее важно то, что он реализует интерфейс `Transferable`, поэтому эти рисунки (отрезки линий, scribbles) могут передаваться от одного приложения другому.

Класс `Scribble` фактически поддерживает передачу данных с помощью двух форматов данных. Он определяет пользовательский формат, передающий сериализованные экземпляры `Scribble`. Однако, как было отмечено в начале главы, такие пользовательские форматы не могут применяться для передачи данных между независимыми виртуальными машинами (по крайней мере, в реализации Java от фирмы Sun). Следовательно, класс `Scribble` позволяет передавать себя также и в виде объекта `String` и определяет методы для преобразования рисунков (отрезков линий) в формат строки и обратно. Заметим, что в примере 13.2 не демонстрируется передача данных методами «вырезание и вставка» или «перетаскивание», в нем просто определяется инфраструктура передачи данных, используемая в последующих примерах, иллюстрирующих механизмы вырезания и вставки, а также перетаскивания. Он также заслуживает изучения как пользовательская реализация `Shape`.

Пример 13.2. Scribble.java

```
package com.davidflanagan.examples.datatransfer;
import java.awt.*;
import java.awt.geom.*;
import java.awt.datatransfer.*;
import java.io.Serializable;
import java.util.StringTokenizer;

/**
 * Данный класс представляет рисунок, составленный из произвольного числа
 * "полилиний". Каждая полилиния представляет собой набор соединенных между
 * собой отрезков. Рисунок создается при помощи серии вызовов методов
 * moveto() и lineto(). moveto() указывает начальную точку новой полилинии,
 * а lineto() добавляет к концу текущей полилинии новую точку.
 *
 * Данный класс реализует интерфейс Shape, это значит, что он может быть
 * отображен с помощью программного графического интерфейса Java2D.
 *
 * Кроме этого, он реализует интерфейс Transferable, то есть он легко может
 * быть использован в операциях вырезания-вставки и перетаскивания.
 * Он определяет пользовательский формат DataFlavorscribbleDataFlavor,
 * который передает объекты Scribble в виде Java-объектов. Однако он также
```

```
 * поддерживает механизмы вырезания-вставки и перетаскивания, основанные
 * на переносимом строковом представлении фрагмента.
 * Методы toString() и parse() записывают и читают этот строковый формат.
 **/
public class Scribble implements Shape, Transferable, Serializable, Cloneable
{
  protected double[] points = new double[64]; // Данные рисунка
  protected int numPoints = 0;                // Текущее количество точек
  double maxX = Double.NEGATIVE_INFINITY;     // Ограничивающий прямоугольник
  double maxY = Double.NEGATIVE_INFINITY;
  double minX = Double.POSITIVE_INFINITY;
  double minY = Double.POSITIVE_INFINITY;

  /**
   * Начинаем новый многоугольник с координатами (x,y).
   * Обратите внимание на использование константы Double.NaN в массиве точек
   * для отметки начала новой полилинии.
   **/
  public void moveto(double x, double y) {
    if (numPoints + 3 > points.length) reallocate();
    // Отмечаем ее как начало новой линии
    points[numPoints++] = Double.NaN;
    // Окончание этого метода просто похоже на lineto();
    lineto(x, y);
  }

  /**
   * Добавляем к концу текущей полилинии точку (x,y)
   **/
  public void lineto(double x, double y) {
    if (numPoints + 2 > points.length) reallocate();
    points[numPoints++] = x;
    points[numPoints++] = y;

    // Смотрим, не выходит ли точка за пределы ограничивающего прямоугольника
    if (x > maxX) maxX = x;
    if (x < minX) minX = x;
    if (y > maxY) maxY = y;
    if (y < minY) minY = y;
  }

  /**
   * Добавляем рисунок (Scribble) "s" к данному рисунку Scribble
   **/
  public void append(Scribble s) {
    int n = numPoints + s.numPoints;
    double[] newpoints = new double[n];
    System.arraycopy(points, 0, newpoints, 0, numPoints);
    System.arraycopy(s.points, 0, newpoints, numPoints, s.numPoints);
    points = newpoints;
    numPoints = n;
    minX = Math.min(minX, s.minX);
    maxX = Math.max(maxX, s.maxX);
```

```
    minY = Math.min(minY, s.minY);
    maxY = Math.max(maxY, s.maxY);
  }

  /**
   * Преобразуем координаты всех точек рисунка Scribble с помощью x,y
   **/
  public void translate(double x, double y) {
    for(int i = 0; i < numPoints; i++) {
      if (Double.isNaN(points[i])) continue;
      points[i++] += x;
      points[i] += y;
    }
    minX += x; maxX += x;
    minY += y; maxY += y;
  }

  /** Внутренний метод для выделения дополнительного места в массиве данных */
  protected void reallocate() {
    double[] newpoints = new double[points.length * 2];
    System.arraycopy(points, 0, newpoints, 0, numPoints);
    points = newpoints;
  }

  /** Копируем объект Scribble и его внутренний массив данных */
  public Object clone() {
    try {
      Scribble s = (Scribble) super.clone(); // Делаем копию всех полей
      s.points = (double[]) points.clone();  // Копируем весь массив
      return s;
    }
    catch (CloneNotSupportedException e) {   // Этого никогда не должно
                                             // случиться
      return this;
    }
  }

  /** Конвертируем данные рисунка в текстовый формат */
  public String toString() {
    StringBuffer b = new StringBuffer();
    for(int i = 0; i < numPoints; i++) {
      if (Double.isNaN(points[i])) {
        b.append("m ");
      }
      else {
        b.append(points[i]);
        b.append(' ');
      }
    }
    return b.toString();
  }

  /**
```

```
 * Создаем новый объект Scribble и инициализируем его путем
 * разбора строки, содержащей координаты в формате,
 * сгенерированном методом toString()
 **/
public static Scribble parse(String s) throws NumberFormatException {
  StringTokenizer st = new StringTokenizer(s);
  Scribble scribble = new Scribble();
  while(st.hasMoreTokens()) {
    String t = st.nextToken();
    if (t.charAt(0) == 'm') {
      scribble.moveto(Double.parseDouble(st.nextToken()),
              Double.parseDouble(st.nextToken()));
    }
    else {
      scribble.lineto(Double.parseDouble(t),
              Double.parseDouble(st.nextToken()));
    }
  }
  return scribble;
}

// ===== Следующие методы реализуют интерфейс Shape =====

/** Возвращаем ограничивающий прямоугольник объекта Shape */
public Rectangle getBounds() {
  return new Rectangle((int)(minX-0.5f), (int)(minY-0.5f),
             (int)(maxX-minX+0.5f), (int)(maxY-minY+0.5f));
}

/** Возвращаемый ограничивающий прямоугольник объекта Shape */
public Rectangle2D getBounds2D() {
  return new Rectangle2D.Double(minX, minY, maxX-minX, maxY-minY);
}

/** Наша фигура является незамкнутой кривой, поэтому она
    не может ничего содержать */
public boolean contains(Point2D p) { return false; }
public boolean contains(Rectangle2D r) { return false; }
public boolean contains(double x, double y) { return false; }
public boolean contains(double x, double y, double w, double h) {
  return false;
}

/**
 * Определяем, пересекается ли рисунок с указанным прямоугольником,
 * проверяя по отдельности каждый отрезок линии
 **/
public boolean intersects(Rectangle2D r) {
  if (numPoints < 4) return false;
  int i = 0;
  double x1, y1, x2 = 0.0, y2 = 0.0;
  while(i < numPoints) {
    if (Double.isNaN(points[i])) { // Если мы начинаем новую линию
```

```
      i++;        //  Пропускаем NaN
      x2 = points[i++];
      y2 = points[i++];
    }
    else {
      x1 = x2;
      y1 = y2;
      x2 = points[i++];
      y2 = points[i++];
      if (r.intersectsLine(x1, y1, x2, y2)) return true;
    }
  }

  return false;
}

/** Проверка на пересечение с помощью вызова предыдущего метода  */
public boolean intersects(double x, double y, double w, double h){
  return intersects(new Rectangle2D.Double(x,y,w,h));
}

/**
 * Возвращаем объект PathIterator, сообщающий Java2D,
 * как нужно изображать данный рисунок
 **/
public PathIterator getPathIterator(AffineTransform at) {
  return new ScribbleIterator(at);
}

/**
 * Возвращаем PathIterator, не содержащий кривых. Наш никогда не содержит.
 **/
public PathIterator getPathIterator(AffineTransform at, double flatness) {
  return getPathIterator(at);
}

/**
 * Этот внутренний класс реализует интерфейс PathIterator для описания
 * формы рисунка. Такие, как Scribble, состоят из произвольного
 * количества точек и линий, мы просто возвращаем их координаты.
 **/
public class ScribbleIterator implements PathIterator {
  protected int i = 0;         // Позиция в массиве
  protected AffineTransform transform;

  public ScribbleIterator(AffineTransform transform) {
    this.transform = transform;
  }

  /** Как определить внутреннюю и внешнюю части этой формы */
  public int getWindingRule() { return PathIterator.WIND_NON_ZERO; }

  /** Достигли ли мы конца пути рисунка? */
  public boolean isDone() { return i >= numPoints; }
```

```
  /** Перемещаемся к следующему отрезку пути */
  public void next() {
    if (Double.isNaN(points[i])) i += 3;
    else i += 2;
  }

  /**
   * Получаем координаты текущей точки и линии в виде массива float
   **/
  public int currentSegment(float[] coords) {
    int retval;
    if (Double.isNaN(points[i])) {    // Если это точка
      coords[0] = (float)points[i+1];
      coords[1] = (float)points[i+2];
      retval = SEG_MOVETO;
    }
    else {
      coords[0] = (float)points[i];
      coords[1] = (float)points[i+1];
      retval = SEG_LINETO;
    }

    // Если было задано преобразование, применяем его для всех координат
    if (transform != null)
        transform.transform(coords, 0, coords, 0,1);

    return retval;
  }

  /**
   * Получаем координаты точек и линий в виде массива double
   **/
  public int currentSegment(double[] coords) {
    int retval;
    if (Double.isNaN(points[i])) {
      coords[0] = points[i+1];
      coords[1] = points[i+2];
      retval = SEG_MOVETO;
    }
    else {
      coords[0] = points[i];
      coords[1] = points[i+1];
      retval = SEG_LINETO;
    }
    if (transform != null) transform.transform(coords, 0, coords, 0,1);
    return retval;
  }
}

//========== Следующие методы реализуют интерфейс Transferable ==========

// Это пользовательский DataFlavor для объектов
// Scribble public static DataFlavor scribbleDataFlavor =
  new DataFlavor(Scribble.class, "Scribble");
```

```
    // Это список форматов, с которыми мы умеем работать
    public static DataFlavor[] supportedFlavors = {
      scribbleDataFlavor,
      DataFlavor.stringFlavor
    };

    /** Возвращаем форматы данных, или просто "форматы" (flavors),
        которые мы умеем передавать */
    public DataFlavor[] getTransferDataFlavors() {
      return (DataFlavor[]) supportedFlavors.clone();
    }

    /** Проверяем, поддерживаем ли мы данный формат */
    public boolean isDataFlavorSupported(DataFlavor flavor) {
      return (flavor.equals(scribbleDataFlavor) ||
          flavor.equals(DataFlavor.stringFlavor));
    }

    /**
     * Возвращаем данные рисунка в запрошенном формате или генерируем
     * исключение, если мы не поддерживаем запрошенный формат
     **/
    public Object getTransferData(DataFlavor flavor)
      throws UnsupportedFlavorException
    {
      if (flavor.equals(scribbleDataFlavor)) { return this; }
      else if (flavor.equals(DataFlavor.stringFlavor)) { return toString(); }
      else throw new UnsupportedFlavorException(flavor);
    }
}
```

Вырезание и вставка рисунков

В примере 13.3 показано Swing-приложение, позволяющее пользователю рисовать мышью линии, а затем передавать их с помощью механизма вырезания и вставки. Для хранения, отображения и передачи рисунка в нем используется класс `Scribble`, определенный в примере 13.2. Возможности по вырезанию и вставке реализованы в методах `cut()`, `copy()` и `paste()`, которые доступны пользователю через всплывающие меню. Обратите внимание, как метод `paste()` пытается передавать данные сначала с помощью пользовательского формата данных `Scribble`, а в случае неудачи снова делает попытку, используя предопределенный строковый формат данных. Чтобы протестировать программу, запустите две ее копии и передавайте фрагменты копий в обоих направлениях.

Пример 13.3. ScribbleCutAndPaste.java

```
package com.davidflanagan.examples.datatransfer;
import java.awt.*;
import java.awt.event.*;
```

```
import javax.swing.*;
import java.awt.datatransfer.*; // Clipboard, Transferable, DataFlavor и др.

/**
 * Этот компонент позволяет пользователю рисовать изображения в окне,
 * а также вырезать и вставлять их в приложения. Он хранит координаты мыши
 * в объекте Scribble, который используется при отображении рисунка,
 * а также для передачи рисунка в буфер обмена и обратно.
 * Компонент JPopupMenu предоставляет доступ к командам
 * cut (вырезать), copy (скопировать) и paste (вставить).
 **/
public class ScribbleCutAndPaste extends JComponent
  implements ActionListener, ClipboardOwner
{
  Stroke linestyle = new BasicStroke(3.0f); // Рисуем широкими линиями
  Scribble scribble = new Scribble();       // Сохраняем наш рисунок
  Scribble selection;           // Копирование рисунка через вырезание
  JPopupMenu popup;             // Меню для команд вырезания и вставки

  public ScribbleCutAndPaste() {
    // Создаем "всплывающее" меню
    String[] labels = new String[] {   "Clear", "Cut", "Copy", "Paste" };
    String[] commands = new String[] { "clear", "cut", "copy", "paste" };
    popup = new JPopupMenu();                   // Создаем меню
    popup.setLabel("Edit");
    for(int i = 0; i < labels.length; i++) {
      JMenuItem mi = new JMenuItem(labels[i]); // Создаем элемент меню
      mi.setActionCommand(commands[i]);    // Задаем для него действие
      mi.addActionListener(this);          // и слушатель этого действия
      popup.add(mi);                       // Добавляем элемент к меню
    }
    // Наконец, регистрируем меню в компоненте, для которого
    // оно будет появляться
    this.add(popup);

    // Добавляем слушатели событий для выполнения отрисовки и обработки меню
    addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent e) {
          if (e.isPopupTrigger())
            popup.show((Component)e.getSource(),
                  e.getX(), e.getY());
          else
            scribble.moveto(e.getX(), e.getY()); // Начинаем новую линию
        }
      });

    addMouseMotionListener(new MouseMotionAdapter() {
        public void mouseDragged(MouseEvent e) {
          // Если это не кнопка 1 мыши, игнорируем ее
          if ((e.getModifiers() & InputEvent.BUTTON1_MASK) == 0)
            return;
          scribble.lineto(e.getX(), e.getY());  // Добавляем линию
          repaint();
```

```
      }
    });
}
/**
 * Рисуем компонент. Этот метод основан на объекте Scribble,
 * реализующем Shape.
 **/
public void paintComponent(Graphics g) {
  super.paintComponent(g);
  Graphics2D g2 = (Graphics2D) g;
  g2.setStroke(linestyle);  // Задаем широкие линии
  g2.draw(scribble);        // Рисуем scribble (рисунок)
}

/** Это метод интерфейса ActionListener, вызываемый
    элементами всплывающего меню */
public void actionPerformed(ActionEvent event) {
  String command = event.getActionCommand();
  if (command.equals("clear")) clear();
  else if (command.equals("cut")) cut();
  else if (command.equals("copy")) copy();
  else if (command.equals("paste")) paste();
}

/** Этот метод очищает рисунок. Вызывается всплывающим меню */
void clear() {
  scribble = new Scribble();  // Получаем новый пустой рисунок
  repaint();                  // и перерисовываем все.
}

/**
 * Делаем копию текущего рисунка и помещаем ее в буфер обмена.
 * Мы можем сделать это, так как Scribble реализует интерфейс Transferable.
 * Пользователь вызывает данный метод с помощью всплывающего меню.
 **/
public void copy() {
  // Получаем системный буфер обмена
  Clipboard c = this.getToolkit().getSystemClipboard();

  // Делаем копию объекта Scribble для передачи в буфер обмена
  selection = (Scribble) scribble.clone();

  // Помещаем копию в буфер обмена
  c.setContents(selection, // Что поместить в буфер обмена
         this);            // Кого уведомить о том, что "это"
                           // там больше не находится
}

/**
 * Команда cut (вырезать) похожа на команду copy (копировать),
 * за исключением того, что мы удаляем текущий рисунок после его
 * копирования в буфер обмена.
 **/
public void cut() {
```

```
    copy();
    clear();
  }
  /**
   * Пользователь вызывает этот метод через всплывающее меню.
   * Сначала запрашиваем объект Transferable, находящийся в системном
   * буфере обмена. Затем запрашиваем у этого объекта Transferable
   * данные рисунка, который он представляет. Пытаемся использовать оба
   * формата данных, поддерживаемых классом Scribble. Если это не работает,
   * выдаем звуковой сигнал, чтобы сообщить пользователю о неудаче операции.
   **/
  public void paste() {
    Clipboard c = this.getToolkit().getSystemClipboard(); // Получаем буфер
                                                          // обмена
    Transferable t = c.getContents(this);      // Получаем его содержимое

    // Сейчас пытаемся получить объект Scribble
    Scribble pastedScribble = null;
    try {
      pastedScribble =
(Scribble)t.getTransferData(Scribble.scribbleDataFlavor);
    }
    catch (Exception e) { // UnsupportedFlavor, NullPointer и т.п.
      // Если это не работает, пытаемся вместо этого запросить строку.
      try {
        String s = (String)t.getTransferData(DataFlavor.stringFlavor);
        // Мы получили строку и пытаемся преобразовать ее в объект Scribble
        pastedScribble = Scribble.parse(s);
      }
      catch (Exception e2) { // UnsupportedFlavor, NumberFormat и т.п.
        // Если мы не смогли получить и преобразовать строку,
        // прекращаем работу.
       this.getToolkit().beep();  // Сообщаем пользователю об ошибке вставки
        return;
      }
    }

    // Если мы оказались здесь, значит, мы получили объект Scribble из буфера
    // обмена. Добавляем его к текущему рисунку и запрашиваем перерисовку.
    repaint();
  }

  /**
   * Этот метод реализует интерфейс ClipboardOwner. Мы указываем
   * ClipboardOwner, когда копируем рисунок в буфер обмена.
   * Этот метод будет вызван, когда в буфер обмена будет скопировано
   * что-то другое, и оно вытолкнет оттуда наши данные. Когда будет вызван
   * этот метод, нам больше не нужно поддерживать нашу копию объекта Scribble,
   * так как он больше не доступен для вставки. Обычно компонент выделяет
   * выбранный объект, пока тот находится в буфере обмена, и использует
   * этот метод для снятия выделения с объекта, когда он больше
   * не находится в буфере обмена.
```

```
     **/
    public void lostOwnership(Clipboard c, Transferable t) {
      selection = null;
    }
    /** Простой метод main для тестирования класса. */
    public static void main(String[] args) {
      JFrame frame = new JFrame("ScribbleCutAndPaste");
      ScribbleCutAndPaste s = new ScribbleCutAndPaste();
      frame.getContentPane().add(s, BorderLayout.CENTER);
      frame.setSize(400, 400);
      frame.setVisible(true);
    }
  }
```

Перетаскивание рисунков

В Java 1.2 была введена поддержка обмена данными методом «перетаскивание» (drag-and-drop). Программный интерфейс для этого механизма находится в пакете `java.awt.dnd` и основан на той же архитектуре `DataFlavor` и `Transferable`, что и механизм вырезания и вставки. В примере 13.4 приведена программа, позволяющая пользователю создать изображение в режиме «рисование» и перетащить его в режиме «перетаскивание».

Программный интерфейс для механизма перетаскивания значительно сложнее интерфейса вырезания и вставки, что отразилось на длине этого примера. Ключевыми интерфейсами являются `DragGestureListener`, который начинает новое перемещение, и `DragSourceListener` и `DropTargetListener`, которые уведомляют источник и приемник перетаскивания о некоторых важных событиях, возникающих во время перетаскивания. В примере реализованы все три интерфейса. Обратите внимание на то, что эти интерфейсы определяют методы с похожими именами и используют запутывающее разнообразие типов событий. Более детальное описание программного интерфейса перетаскивания можно найти в книге «Java Foundation Classes in a Nutshell».

Во время изучения данного примера вам следует уделить особое внимание методам `dragGestureRecognized()`, указывающему место инициализации операции перетаскивания, и `drop()`, обозначающему действительное место передачи данных от источника приемнику. В этом примере используется список `List` объектов `Scribble`, каждый из которых представляет набор соединенных друг с другом отрезков линий. Этим он отличается от примера 13.3, в котором для представления всего рисунка использовался единственный объект `Scribble`, включающий соединенные и отдельные отрезки.

Для того чтобы протестировать программу, запустите одну ее копию и нарисуйте мышью несколько линий. Затем щелкните на кнопке **Drag** и перетащите мышью несколько этих линий. Затем запустите еще од-

ну копию программы и перетащите линии из одной программы в другую. Отметим, что программный интерфейс Java перетаскивания может взаимодействовать с «родной» для вашего компьютера системой перетаскивания. Он использует ту же привязку мыши и клавиатуры, что и «родное» приложение. Обычно при операции перетаскивания по умолчанию выполняется перемещение рисунка. Но, удерживая соответствующую клавишу-модификатор, вы можете превратить ее в операцию копирования.

Пример 13.4. ScribbleDragAndDrop.java

```
package com.davidflanagan.examples.datatransfer;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import java.awt.datatransfer.*; // Clipboard, Transferable, DataFlavor и др.
import java.awt.dnd.*;
import java.util.ArrayList;

/**
 * Этот компонент может работать в двух режимах. В режиме "рисование"
 * он позволяет чертить мышью линии. В режиме "перетаскивание"
 * он позволяет пользователю перетаскивать мышью этот рисунок.
 * Независимо от режима он может принимать рисунки, перетащенные
 * из других приложений.
 **/
public class ScribbleDragAndDrop extends JComponent
  implements DragGestureListener,  // Для распознавания начала перетаскивания
        DragSourceListener,   // Для обработки событий источника перетаскивания
        DropTargetListener,   // Для обработки событий приемника перетаскивания
         MouseListener,       // Для обработки щелчков мыши
         MouseMotionListener  // Для обработки перетаскиваний мыши
{
  ArrayList scribbles = new ArrayList(); // Список рисуемых объектов Scribble
  Scribble currentScribble;              // Рисуемый объект
  Scribble beingDragged;                 // Перетаскиваемый объект
  DragSource dragSource;                 // Центральный объект перетаскивания
  boolean dragMode;                      // Рисуем или перетаскиваем?

  // Некоторые из используемых констант
  static final int LINEWIDTH = 3;
  static final BasicStroke linestyle = new BasicStroke(LINEWIDTH);
  static final Border normalBorder = new BevelBorder(BevelBorder.LOWERED);
  static final Border dropBorder = new BevelBorder(BevelBorder.RAISED);

  /** Конструктор: готовим все необходимое для перетаскивания */
  public ScribbleDragAndDrop() {
    // Используем принятую рамку, установленную по умолчанию.
    // Мы изменим ее в процессе перетаскивания.
    setBorder(normalBorder);

    // Регистрируем слушатели для обработки рисования
```

```
    addMouseListener(this);
    addMouseMotionListener(this);

    // Создаем DragSource и DragGestureRecognizer для прослушивания
    // событий перетаскивания. DragGestureRecognizer будет сообщать
    // объекту DragGestureListener о том, что пользователь делает
    // попытку перетащить объект.
    dragSource = DragSource.getDefaultDragSource();
    dragSource.createDefaultDragGestureRecognizer(
     this,                              // Какой компонент?
     DnDConstants.ACTION_COPY_OR_MOVE, // Какой тип перетаскивания?
     this);                             // слушатель

    // Создаем и настраиваем DropTarget, который будет обрабатывать события
    // по перетаскиванию и отпусканию объектов над этим компонентом
    // и будет посылать уведомления объекту DropTargetListener
    DropTarget dropTarget = new DropTarget(
              this,                    // Наблюдаемый компонент
              this);                   // Слушатель для отправки уведомлений
    this.setDropTarget(dropTarget);    // Сообщаем о нем компоненту
  }

  /**
   * Компонент отображает себя путем отрисовки каждого объекта Scribble.
   **/
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;
    g2.setStroke(linestyle);  // Устанавливаем широкие линии

    int numScribbles = scribbles.size();
    for(int i = 0; i < numScribbles; i++) {
      Scribble s = (Scribble)scribbles.get(i);
      g2.draw(s);              // Чертим рисунок
    }
  }

  public void setDragMode(boolean dragMode) {
    this.dragMode = dragMode;
  }
  public boolean getDragMode() { return dragMode; }

  /**
   * Этот и следующие четыре метода относятся к интерфейсу MouseListener.
   * Если мы находимся в режиме "рисование", этот метод обрабатывает события
   * нажатия кнопки мыши и начинает новый рисунок.
   **/
  public void mousePressed(MouseEvent e) {
    if (dragMode) return;
    currentScribble = new Scribble();
    scribbles.add(currentScribble);
    currentScribble.moveto(e.getX(), e.getY());
  }
  public void mouseReleased(MouseEvent e) {}
```

```
public void mouseClicked(MouseEvent e) {}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}

/**
* Этот метод и расположенный ниже метод mouseMoved() относятся к интерфейсу
* MouseMotionListener. Если мы находимся в режиме "рисование", этот метод
* добавляет к текущему фрагменту новую точку и запрашивает перерисовку.
**/
public void mouseDragged(MouseEvent e) {
  if (dragMode) return;
  currentScribble.lineto(e.getX(), e.getY());
  repaint();
}
public void mouseMoved(MouseEvent e) {}

/**
 * Этот метод реализует интерфейс DragGestureListener. Он будет вызван
 * тогда, когда DragGestureRecognizer решит, что пользователь начал
 * операцию перетаскивания. Если мы не в режиме "рисование",
 * данный метод попытается выяснить, какой объект Scribble перетаскивается,
 * и инициализирует операцию перетаскивания для этого объекта.
 **/
public void dragGestureRecognized(DragGestureEvent e) {
  // Не перетаскиваем, если мы в режиме "рисование"
  if (!dragMode) return;

  // Выясняем, где началось перетаскивание
  MouseEvent inputEvent = (MouseEvent) e.getTriggerEvent();
  int x = inputEvent.getX();
  int y = inputEvent.getY();

  // Выясняем, над каким рисунком была нажата кнопка мыши, путем создания
  // в этой точке небольшого прямоугольника и проверки на пересечение.
  // Проходим в цикле все рисунки.
  Rectangle r = new Rectangle (x-LINEWIDTH, y-LINEWIDTH,
                LINEWIDTH*2, LINEWIDTH*2);
  int numScribbles = scribbles.size();
  for(int i = 0; i < numScribbles; i++) { // Проходим в цикле все рисунки
    Scribble s = (Scribble) scribbles.get(i);
    if (s.intersects(r)) {
      // Пользователь начал перетаскивание на этом рисунке,
      // поэтому начинаем перемещать его. Прежде всего, запоминаем,
      // какой рисунок перетаскивается, чтобы позднее мы могли
      // удалить его (если это перемещение, а не копирование)
       beingDragged = s;

      // Затем создаем копию, которая будет перетаскиваться
      Scribble dragScribble = (Scribble) s.clone();
      // Устанавливаем точку отсчета в место,
      // где пользователь нажал кнопку мыши.
      dragScribble.translate(-x, -y);
```

```
        // Устанавливаем форму указателя мыши в зависимости от типа
        // инициализированного пользователем перетаскивания
        Cursor cursor;
        switch(e.getDragAction()) {
        case DnDConstants.ACTION_COPY:
          cursor = DragSource.DefaultCopyDrop;
          break;
        case DnDConstants.ACTION_MOVE:
          cursor = DragSource.DefaultMoveDrop;
          break;
        default:
          return;  // Мы поддерживаем только перемещение и копирование
        }

        // Некоторые системы позволяют нам вместе с указателем перемещать
        // и изображение. Если это так, создаем изображение рисунка.
        if (dragSource.isDragImageSupported()) {
          Rectangle scribbleBox = dragScribble.getBounds();
          Image dragImage = this.createImage(scribbleBox.width,
                                scribbleBox.height);
          Graphics2D g = (Graphics2D)dragImage.getGraphics();
          g.setColor(new Color(0,0,0,0)); // Прозрачный фон
          g.fillRect(0, 0, scribbleBox.width, scribbleBox.height);
          g.setColor(Color.black);
          g.setStroke(linestyle);
            g.translate(-scribbleBox.x, -scribbleBox.y);
          g.draw(dragScribble);
          Point hotspot = new Point(-scribbleBox.x, -scribbleBox.y);

          // Начинаем перетаскивание с использованием изображения.
          e.startDrag(cursor, dragImage, hotspot, dragScribble,this);
        }
        else {
          // Или начинаем перетаскивание без изображения
          e.startDrag(cursor, dragScribble,this);
        }
        // После того как мы начали перетаскивание одного
        // рисунка, прекращаем поиск.
        return;
      }
    }
  }

  /**
   * Этот метод и следующие четыре неиспользуемых метода реализуют
   * интерфейс DragSourceListener. Метод dragDropEnd() вызывается тогда,
   * когда пользователь отпускает перетаскиваемый рисунок.
   * Если отпускание было выполнено успешно, и если пользователь
   * делал перемещение, а не копирование, мы удаляем перетащенный
   * рисунок из списка отображаемых рисунков
   **/
  public void dragDropEnd(DragSourceDropEvent e) {
```

```
  if (!e.getDropSuccess()) return;
  int action = e.getDropAction();
  if (action == DnDConstants.ACTION_MOVE) {
    scribbles.remove(beingDragged);
    beingDragged = null;
    repaint();
  }
}

// Эти методы также являются частью DragSourceListener.
// Они вызываются в характерных точках процесса перетаскивания
// и могут использоваться для реализации графических эффектов,
// таких как изменение вида указателя мыши или изображения.
public void dragEnter(DragSourceDragEvent e) {}
public void dragExit(DragSourceEvent e) {}
public void dropActionChanged(DragSourceDragEvent e) {}
public void dragOver(DragSourceDragEvent e) {}

// Следующие пять методов реализуют интерфейс DropTargetListener

/**
 * Этот метод вызывается, когда пользователь перетаскивает что-либо
 * над нами. Если мы понимаем тип данных перетаскиваемого объекта, то
 * вызываем acceptDrag(), чтобы сообщить системе, что мы можем его принять.
 * Также мы меняем рамку для создания эффекта "перетаскивания под",
 * чтобы сигнализировать о возможности принять этот объект.
   **/
public void dragEnter(DropTargetDragEvent e) {
  if (e.isDataFlavorSupported(Scribble.scribbleDataFlavor)
    ||e.isDataFlavorSupported(DataFlavor.stringFlavor)) {
      e.acceptDrag(DnDConstants.ACTION_COPY_OR_MOVE);
      this.setBorder(dropBorder);
  }
}

/**
 * Пользователь выполняет перетаскивание уже не над нами,
 * поэтому восстанавливаем рамку
 **/
public void dragExit(DropTargetEvent e) {
    this.setBorder(normalBorder);
}

/**
 * Это основной метод интерфейса DropTargetListener.
 * Он вызывается, когда пользователь отпускает что-либо над нами.
 **/
public void drop(DropTargetDropEvent e) {
  // Восстанавливаем первоначальную рамку
  this.setBorder(normalBorder);

  // Сначала проверяем, понимаем ли мы формат данных объекта,
  // который был над нами отпущен. Если мы поддерживаем его форматы
  // данных, принимаем его, иначе – отвергаем.
```

```
    if (e.isDataFlavorSupported(Scribble.scribbleDataFlavor)
      ||e.isDataFlavorSupported(DataFlavor.stringFlavor)) {
      e.acceptDrop(DnDConstants.ACTION_COPY_OR_MOVE);
    }
    else {
      e.rejectDrop();
      return;
    }

    // Мы приняли объект, поэтому сейчас пробуем извлечь
    // его данные из объекта Transferable.
    Transferable t = e.getTransferable(); // Сохраняем данные
    Scribble droppedScribble; // Здесь будет храниться объект Scribble

    // Сначала пытаемся получить данные прямо в виде объекта Scribble
    try {
      droppedScribble = (Scribble)
        t.getTransferData(Scribble.scribbleDataFlavor);
    }
    catch (Exception ex) { // Неизвестный формат, исключение ввода/вывода и др.
      // Если это не сработало, пытаемся получить его
      // в виде строки, а затем проанализировать (parse) ее.
      try {
        String s = (String)
          t.getTransferData(DataFlavor.stringFlavor);
        droppedScribble = Scribble.parse(s);
      }
      catch(Exception ex2) {
        // Если мы все же не смогли извлечь данные,
        // сообщаем системе о нашей неудаче
        e.dropComplete(false);
        return;
      }
    }

    // Если мы дошли до этого места, значит, мы получили объект Scribble
    Point p = e.getLocation();  // Где произошло отпускание
    // Перемещаем его туда
    droppedScribble.translate(p.getX(), p.getY());
    // Вносим в список отображения
    scribbles.add(droppedScribble);
    repaint();                  // Требуем перерисовки
    e.dropComplete(true);       // Сообщаем об успехе!
  }

  // Это неиспользуемые методы интерфейса DropTargetListener
  public void dragOver(DropTargetDragEvent e) {}
  public void dropActionChanged(DropTargetDragEvent e) {}

  /**
   * Метод main(). Создает с помощью этого класса простое
   * приложение. Обратите внимание на кнопки, служащие
   * для переключения между режимами "рисование" и "перетаскивание".
```

```
     **/
    public static void main(String[] args) {
      // Создаем окно и помещаем в него панель для рисования
      JFrame frame = new JFrame("ScribbleDragAndDrop");
      final ScribbleDragAndDrop scribblePane = new ScribbleDragAndDrop();
      frame.getContentPane().add(scribblePane, BorderLayout.CENTER);

      // Создаем две кнопки переключения режимов
      JToolBar toolbar = new JToolBar();
      ButtonGroup group = new ButtonGroup();
      JToggleButton draw = new JToggleButton("Draw");
      JToggleButton drag = new JToggleButton("Drag");
      draw.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
            scribblePane.setDragMode(false);
          }
        });
      drag.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
            scribblePane.setDragMode(true);
          }
        });
      group.add(draw); group.add(drag);
      toolbar.add(draw); toolbar.add(drag);
      frame.getContentPane().add(toolbar, BorderLayout.NORTH);

      // Начинаем в режиме "рисование"
      draw.setSelected(true);
      scribblePane.setDragMode(false);

      // Показываем окно
      frame.setSize(400, 400);
      frame.setVisible(true);
    }
  }
```

Упражнения

13-1. **Изучите класс** `StringSelection`**. Он реализует интерфейс** `Transferable` **для класса** `String` **и облегчает передачу строк между приложениями. Создайте класс** `SimpleSelection`**, реализующий интерфейсы** `Transferable` **и** `ClipboardOwner` **точно так же, как это делает класс** `StringSelection`**. Конструктор вашего класса должен принимать в качестве своего единственного параметра произвольный сериализуемый объект; этот объект является «выделением», которое необходимо передать. Создайте класс** `DataFlavor` **на основе класса представления для этого объекта (см.** `Object.getClass()`**) и реализуйте методы интерфейса** `Transferable`**, используя разработанный вами** `DataFlavor`**.**

Теперь модифицируйте пример 13.1 так, чтобы он использовал `SimpleSelection` для передачи объектов `Integer` методом вырезания и вставки. Обратите внимание, что `SimpleSelection` будет, возможно, работать только в случае вырезания и вставки внутри одного приложения.

13-2. Доработайте пример 13.4 так, чтобы он кроме механизма перетаскивания поддерживал вырезание и вставку. Предоставьте пользователю возможность выделять рисунок двойным щелчком мыши (см. `MouseEvent.getClickCount()`). Разрешите пользователю вставлять рисунок из буфера обмена при помощи пункта меню, мыши или клавиатуры на ваш выбор. Когда пользователь делает двойной щелчок на рисунке, выделите этот рисунок другим цветом и снимите выделение только тогда, когда в буфер обмена будет помещено что-то другое.

Глава 14

JavaBeans

Программный интерфейс JavaBeans предоставляет среду для разработки многократно используемых, встраиваемых, модульных программных компонентов. Спецификация JavaBeans дает следующее определение компонента (bean): «многократно используемый программный компонент, которым можно манипулировать в визуальных средах разработки». Вы видите, что это достаточно свободное определение; компоненты могут принимать самые разные формы. На самом простом уровне компонентами JavaBeans являются все отдельные графические компоненты, тогда как на гораздо более высоком уровне в качестве компонента может также функционировать встраиваемое приложение с электронными таблицами. Однако большинство компонентов, скорее всего, находятся где-то между этими двумя крайностями.

Одной из целей модели JavaBeans является обеспечение взаимодействия с аналогичными компонентными средами. Так, например, обычная Windows-программа может с помощью соответствующего моста или компонента-обертки пользоваться компонентом Java так, как будто бы он является компонентом COM или ActiveX. Однако детали этого взаимодействия выходят за рамки этой главы.

Компоненты могут использоваться на трех уровнях, тремя разными категориями программистов:

- Если вы разрабатываете редакторы графического интерфейса пользователя (GUI), конструкторы приложений (application builders) или другие контейнерные («beanbox») средства, интерфейс JavaBeans вам будет нужен для манипулирования компонентами внутри этих инструментальных средств. *beanbox* (контейнер компонентов) – это название простой программы управления компонентами, поставляемой фирмой Sun в составе своего пакета JavaBeans Development

Kit™ (BDK).[1] Это полезный термин, и я буду использовать его для описания всех типов средств визуальной разработки и конструкторов приложений, которые манипулируют компонентами.

- Если вы непосредственно занимаетесь разработкой компонентов, интерфейс JavaBeans вам будет нужен для написания кода, которым может пользоваться любой подходящий контейнер компонентов.
- Если вы пишете приложения, в которых используются компоненты, разработанные другими программистами, или используете контейнер компонентов для сборки этих компонентов в одно приложение, вам в действительности не понадобится знакомство с интерфейсом JavaBeans. Вам потребуется только знание документации по отдельным используемым вами компонентам.

В этой главе объясняется, как нужно использовать программный интерфейс JavaBeans на втором уровне, или, другими словами, в ней описано, как создавать компоненты. Рассмотрены следующие темы:

- Основные понятия и термины, связанные с компонентами
- Требования для простейших компонентов
- Упаковка компонентов в файлы JAR
- Предоставление дополнительной информации о компонентах с помощью класса `BeanInfo`
- Определение редакторов свойств, позволяющих настраивать свойства компонентов
- Определение настройщиков, позволяющих настраивать параметры всего компонента

Обратите внимание на то, что в примерах этой главы рассматриваются AWT-компоненты для использования с Java 1.0 и 1.1. Хотя в настоящее время Swing-компоненты используются более широко, чем AWT-компоненты, достаточно много примеров Swing-компонентов было приведено в главе 10 «Графические интерфейсы пользователя (GUI)». Таким образом, примеры этой главы несут двойную пользу, демонстрируя как AWT, так и JavaBeans.

Основы компонентов

Мы начнем наше рассмотрение компонентов с некоторых базовых понятий и терминов. Любой объект, удовлетворяющий определенным базовым правилам и соглашениям об именах[2], может считаться ком-

[1] Этот BDK можно загрузить с сайта *http://java.sun.com/beans*, если у вас нет никакой среды разработки, ориентированной на компоненты JavaBeans.

[2] Описание этих правил и соглашений об именах см. в главе 6 книги «Java in a Nutshell» («Java. Справочник», 4-е издание, Символ-Плюс, 2003). Данный обзор JavaBeans – это выдержки из упоминаемой главы.

понентом JavaBeans. Не существует никакого класса Bean, который все компоненты должны были бы иметь в качестве базового. Многие компоненты являются AWT-компонентами, но также возможно, и часто полезно, создавать «невидимые» компоненты, не имеющие видимого представления. (Однако тот факт, что компонент не имеет видимого представления в конечном приложении, не означает, что им нельзя визуально манипулировать контейнерными средствами.)

Любой компонент экспортирует свойства, события и методы. *Свойство* (*property*) – это часть внутреннего состояния компонента, которую можно программно устанавливать и опрашивать, обычно через стандартную пару методов доступа get и set. Компонент может генерировать *события* (*events*) таким же образом, как AWT-компонент, например Button, генерирует события ActionEvent. Интерфейс JavaBeans использует ту же модель событий (фактически он определяет модель событий), которая используется в графических пользовательских интерфейсах Swing и AWT в Java 1.1 и выше. Полное описание этой модели см. в главе 10. Компонент определяет событие путем предоставления методов для добавления и удаления объектов-слушателей событий из списка слушателей, заинтересованных в данном событии. И наконец, *методы* (*methods*), экспортируемые компонентом, являются простыми открытыми (public) методами, определенными в компоненте, за исключением методов, используемых для установки (set) и чтения (get) свойств и регистрации и удаления слушателей событий.

Кроме обычных, только что описанных типов свойств интерфейс JavaBeans обеспечивает поддержку индексных (indexed), связанных (bound) и ограниченных (constrained) свойств. *Индексное свойство* – это любое свойство, имеющее значение типа «массив» и для которого компонент предоставляет методы для извлечения и установки отдельных компонентов этого массива, а также методы для считывания и записи массива целиком. *Связанное свойство* – это свойство, которое рассылает уведомление при изменении своего значения, тогда как *ограниченное свойство* – это то, которое рассылает уведомление при изменении своего значения и позволяет слушателям запретить это изменение.

Из-за того что Java разрешает загружать классы динамически, контейнерные программы могут загружать неизвестные им классы. Контейнерные средства определяют поддерживаемые компонентом свойства, события и методы при помощи механизма интроспекции (introspection), основанном на механизме отражения (reflection) java.lang.reflect, предназначенном для получения информации о членах класса. Кроме этого, компонент может предоставлять вспомогательный класс BeanInfo, поставляющий дополнительную информацию о компоненте. Класс BeanInfo предоставляет эту дополнительную информацию в виде нескольких объектов FeatureDescriptor, каждый из которых описывает какое-либо отдельное свойство компонента. У FeatureDescriptor есть не-

сколько подклассов: BeanDescriptor, PropertyDescriptor, IndexedPropertyDescriptor, EventSetDescriptor, MethodDescriptor и ParameterDescriptor.

Основной задачей контейнерного приложения является предоставление пользователю возможности настройки компонента путем установки значений свойств. Контейнеры определяют редакторы свойств для широко используемых типов свойств, таких как числа, строки, шрифты и цвета. Однако если у компонента есть свойство более сложного типа, для него может потребоваться создание класса ProprtyEditor, с помощью которого контейнер может предоставить пользователю возможность задать значение этого свойства.

Кроме этого, для сложных компонентов механизм настройки при помощи свойств, предоставляемый контейнером, может оказаться недостаточным. Такие компоненты могут определить класс Customizer, который создает графический интерфейс, позволяющий пользователю конфигурировать компонент некоторым удобным для него образом. Особенно сложные компоненты могут даже определять настройщики, которые выступают в качестве мастеров («wizard»), которые проводят пользователя через процесс пошаговой настройки.

Простой компонент

Как было отмечено выше, все Swing- и AWT-компоненты могут функционировать как компоненты JavaBeans. Когда вы пишете пользовательский графический компонент, несложно сделать так, чтобы он мог действовать также в качестве JavaBeans-компонента. В примере 14.1 показано определение пользовательского JavaBeans-компонента MultiLineLabel, отображающего одну или несколько строк статического текста. Это то, чего не может делать AWT-компонент Label.

Этот компонент является JavaBeans-компонентом из-за того, что у всех его свойств есть методы доступа get и set. Так как MultiLineLabel никак не реагирует на действия пользователя, он не определяет никаких событий, поэтому ему не нужны никакие методы для регистрации слушателей событий. Кроме этого, MultiLineLabel определяет конструктор без аргументов, поэтому его экземпляры могут быть легко созданы контейнером компонентов.

Пример 14.1. MultiLineLabel.java

```
package com.davidflanagan.examples.beans;
import java.awt.*;
import java.util.*;

/**
 * Пользовательский компонент, отображающий несколько строк
 * текста с заданными отступами и выравниванием. В Java 1.1
 * мы могли бы породить подкласс от класса Component, сделав
 * его «легковесным» компонентом. Но вместо этого мы пытаемся
 * сделать этот компонент совместимым с Java 1.0.
```

```
 * Это означает, что при компиляции в Java 1.1 или выше вы
 * увидите предупреждения об использовании нерекомендуемых средств.
**/
public class MultiLineLabel extends Canvas {
  // Определяемые пользователем свойства
  protected String label;         // Надпись (метка), без разбиения на строки
  protected int margin_width;     // Левое и правое поля
  protected int margin_height;    // Верхнее и нижнее поля
  protected Alignment alignment;  // Выравнивание текста

  // Вычисляемые значения состояния
  protected int num_lines;            // Количество строк
  protected String[] lines;           // Надпись, разбитая на строки
  protected int[] line_widths;        // Длина каждой строки
  protected int max_width;            // Длина самой длинной строки
  protected int line_height;          // Общая высота шрифта
  protected int line_ascent;          // Высота шрифта над базовой линией
  protected boolean measured = false; // Были ли измерены строки?

  // Пять версий конструктора
  public MultiLineLabel(String label, int margin_width,
             int margin_height, Alignment alignment) {
    this.label = label;         // Запоминаем все свойства
    this.margin_width = margin_width;
    this.margin_height = margin_height;
    this.alignment = alignment;
    newLabel();                 // Разбиваем надпись на строки
  }

  public MultiLineLabel(String label, int margin_width, int margin_height) {
    this(label, margin_width, margin_height, Alignment.LEFT);
  }

  public MultiLineLabel(String label, Alignment alignment) {
    this(label, 10, 10, alignment);
  }

  public MultiLineLabel(String label) { this(label, 10, 10, Alignment.LEFT);}

  public MultiLineLabel() { this(""); }

  // Методы для установки и опроса различных атрибутов компонента.
  // Заметим, что некоторые методы опроса наследованы от базового класса.
  public void setLabel(String label) {
    this.label = label;
    newLabel();           // Разбиваем надпись на строки
    measured = false;     // Отмечаем, что нам нужно измерить строки
    repaint();            // Запрашиваем перерисовку
  }

  public void setFont(Font f) {
    super.setFont(f);     // Сообщает нашему базовому классу новый шрифт
    measured = false;     // Отмечаем, что нам нужно повторно измерить строки
    repaint();            // Запрашиваем перерисовку
```

```
}

public void setForeground(Color c) {
  super.setForeground(c);  // Сообщаем нашему базовому классу о новом цвете
  repaint();               // Запрашиваем перерисовку (размер не изменился)
}

public void setAlignment(Alignment a) { alignment = a; repaint(); }
public void setMarginWidth(int mw) { margin_width = mw; repaint(); }
public void setMarginHeight(int mh) { margin_height = mh; repaint(); }

// Методы получения значений свойств. Заметьте, что getFont(),
// getForeground() и др. унаследованы от базового класса
public String getLabel() { return label; }
public Alignment getAlignment() { return alignment; }
public int getMarginWidth() { return margin_width; }
public int getMarginHeight() { return margin_height; }

/**
 * Этот метод вызывается менеджером компоновки, когда тому
 * потребуется узнать, какой размер желателен.
 * В Java 1.1 предпочтительная версия этого метода - getPreferredSize().
 * Мы пользуемся нерекомендуемой версией для того, чтобы компонент
 * мог взаимодействовать с компонентами версии 1.0.
*/
public Dimension preferredSize() {
  if (!measured) measure();
  return new Dimension(max_width + 2*margin_width,
                       num_lines * line_height + 2*margin_height);
}

/**
 * Этот метод вызывается, когда менеджеру компоновки необходимо
 * узнать минимальный размер необходимого нам места.
 * В Java 1.1 мы могли бы использовать getMinimumSize().
 */
public Dimension minimumSize() { return preferredSize(); }

/**
 * Этот метод рисует компонент. Заметьте, что он обрабатывает границы
 * и выравнивание, но не заботится ни о цвете, ни о шрифте -
 * их установкой в объекте Graphics занимается базовый класс.
 **/
public void paint(Graphics g) {
  int x, y;
  Dimension size = this.size(); // в Java 1.1 использует getSize()
  if (!measured) measure();
  y = line_ascent + (size.height - num_lines * line_height)/2;
  for(int i = 0; i < num_lines; i++, y += line_height) {
    if (alignment == Alignment.LEFT) x = margin_width;
    else if (alignment == Alignment.CENTER)
      x = (size.width - line_widths[i])/2;
    else x = size.width - margin_width - line_widths[i];
    g.drawString(lines[i], x, y);
```

```
    }
  }

  /**
   * Этот внутренний метод разбивает указанную надпись на массив строк.
   * Он использует вспомогательный класс StringTokenizer.
   **/
  protected synchronized void newLabel() {
    StringTokenizer t = new StringTokenizer(label, "\n");
    num_lines = t.countTokens();
    lines = new String[num_lines];
    line_widths = new int[num_lines];
    for(int i = 0; i < num_lines; i++) lines[i] = t.nextToken();
  }

  /**
   * Этот внутренний метод выясняет размеры шрифта, длину каждой
   * строки надписи и размер самой длинной строки.
   **/
  protected synchronized void measure() {
    FontMetrics fm = this.getToolkit().getFontMetrics(this.getFont());
    line_height = fm.getHeight();
    line_ascent = fm.getAscent();
    max_width = 0;
    for(int i = 0; i < num_lines; i++) {
      line_widths[i] = fm.stringWidth(lines[i]);
      if (line_widths[i] > max_width) max_width = line_widths[i];
    }
    measured = true;
  }
}
```

Класс Alignment

В `MultiLineLabel` для определения трех констант выравнивания используется вспомогательный класс с именем `Alignment`. Определение этого класса показано в примере 14.2. В классе определены три константы, содержащие экземпляры самого класса, и объявлен закрытый конструктор, поэтому никакие другие его экземпляры не могут быть созданы. Таким образом, `Alignment` эффективно создает перечислимый (enumerated) тип. Эта полезная технология вовсе не является чем-то особенным для JavaBeans.

Пример 14.2. Alignment.java

```
package com.davidflanagan.examples.beans;

/** Этот класс определяет перечислимый тип с тремя значениями */
public class Alignment {
  /** Этот закрытый конструктор предотвращает создание экземпляров класса */
  private Alignment() {};
  // Следующие три константы являются единственными экземплярами этого класса
```

```
  public static final Alignment LEFT = new Alignment();
  public static final Alignment CENTER = new Alignment();
  public static final Alignment RIGHT = new Alignment();
}
```

Упаковка компонента

Для того чтобы подготовить компонент к использованию в контейнере, вы должны упаковать его вместе с необходимыми ему файлами и ресурсами в файл JAR. (Файлы JAR – это архивы Java (Java archives); вы можете прочитать об утилите *jar* в книге «Java in Nutshell».) Поскольку один компонент может иметь много вспомогательных файлов и поскольку файл JAR может содержать несколько компонентов, в манифесте JAR-файла должно быть описано, какие элементы файла являются компонентами. Файлы JAR создаются при помощи команды *jar* с ключом `c`. Если вместе с ключом `c` используется ключ `m`, *jar* считывает отдельный указанный вами файл манифеста. *jar* использует информацию из указанного вами отдельного файла манифеста при создании полного манифеста для файла JAR. Для указания того, что файл класса является компонентом, вам просто нужно в элемент манифеста файла добавить следующую строчку:

```
Java-Bean: true
```

Для упаковки класса `MultiLineLabel` в файл JAR, прежде всего, создайте «файл-заглушку» манифеста. Создайте файл, скажем, с именем *manifest.stub* и со следующим содержимым:

```
Name: com/davidflanagan/examples/beans/MultiLineLabel.class
Java-Bean: true
```

Обратите внимание, что в системе Windows прямые слэши в файле манифеста не должны меняться на обратные. Формат файла манифеста JAR для разделения каталогов требует использования прямых слэшей независимо от платформы. Создав этот отдельный файл манифеста, вы можете теперь создать файл JAR:

```
% jar cfm MultiLineLabel.jar manifest.stub
com/davidflanagan/examples/beans/MultiLineLabel.class
com/davidflanagan/examples/beans/Alignment.class
```

Заметьте, что это одна длинная команда, разбитая на три строки. Кроме этого, для системы Windows вы должны заменить в этой командной строке прямые слэши на обратные. Если данный компонент требует вспомогательных файлов, вы можете указать их вместе с файлами классов компонента в конце командной строки *jar*.

Установка компонента

Процедура установки компонента зависит от используемого вами контейнерного средства. В случае с утилитой *beanbox*, поставляемой вместе с BDK, все, что вам нужно, – это просто скопировать JAR-файл с компонентом в каталог *jars*, находящийся в каталоге BDK. После того как вы сделаете это, ваш компонент будет появляться в палитре компонентов каждый раз при запуске приложения. Другая возможность состоит в загрузке JAR-файла компонента во время выполнения путем выбора пункта **Load JAR** в меню **File** утилиты *beanbox*.

Более сложный компонент

В примере 14.3 показан еще один компонент – `YesNoPanel`. Этот компонент отображает для пользователя сообщение (при помощи `MultiLineLabel`) и три кнопки. При щелчке мышью на этих кнопках он генерирует событие. Компонент `YesNoPanel` предназначен для использования внутри диалоговых окон, так как он предлагает идеальный способ задать пользователю вопрос «Да/Нет». На рис. 14.1 компонент `YesNoPanel` показан внутри утилиты *beanbox* от фирмы Sun.

Для уведомления объекта `AnswerListener` о том, что пользователь нажал одну из кнопок, компонент `YesNoPanel` использует пользовательский тип `AnswerEvent`. Этот новый класс событий и интерфейс слушателя определены в следующем разделе.

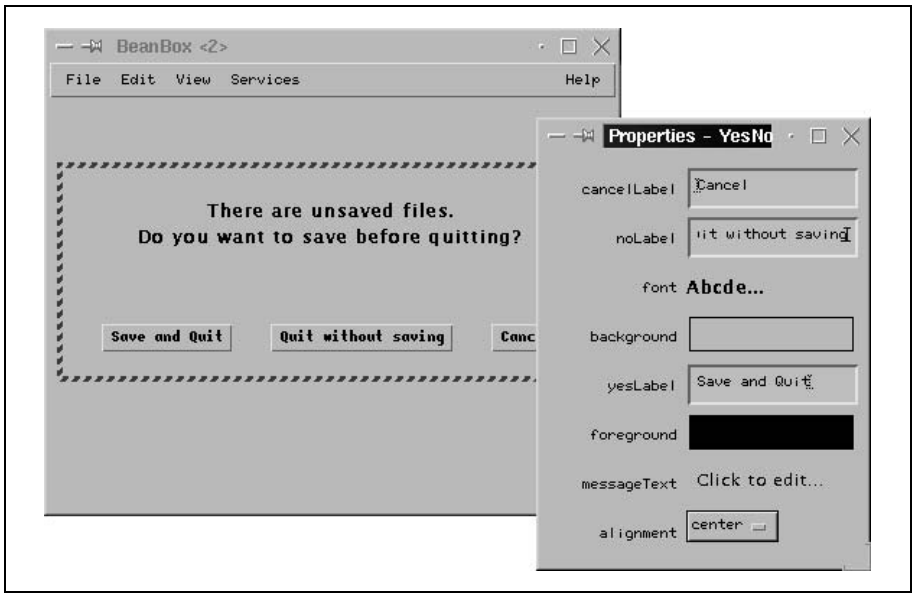


***Рис. 14.1.** Компонент YesNoPanel в утилите beanbox*

Обратите внимание, что компонент `YesNoPanel` не пользуется классами из пакета `java.beans`. Одним из необычных свойств компонентов JavaBean является то, что им обычно не нужно использовать какие-либо классы из этого пакета. Как вы увидите далее в этой главе, трудности для использования этого пакета создают вспомогательные классы, распространяемые вместе с компонентом.

Пример 14.3. YesNoPanel.java

```
package com.davidflanagan.examples.beans;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

/**
 * Этот компонент JavaBean выводит многострочное сообщение и три кнопки.
 * Он активизирует AnswerEvent, когда пользователь
 * нажимает одну из этих кнопок
 **/
public class YesNoPanel extends Panel {
  // Свойства компонента.
  protected String messageText;  // Сообщение для вывода
  protected Alignment alignment; // Тип выравнивания сообщения
  protected String yesLabel;     // Текст для кнопок Yes, No и
  protected String noLabel;      // Cancel
  protected String cancelLabel;

  // Внутренние компоненты панели
  protected MultiLineLabel message;
  protected Button yes, no, cancel;

  /** Безаргументный конструктор компонента со значениями свойств по умолчанию */
  public YesNoPanel() { this("Здесь\nВаше\nСообщение"); }

  public YesNoPanel(String messageText) {
    this(messageText, Alignment.LEFT, "Yes", "No", "Cancel");
  }

  /** Конструктор для использования вашего класса программистами */
  public YesNoPanel(String messageText, Alignment alignment,
           String yesLabel, String noLabel, String cancelLabel)
  {
    // Создаем компоненты для этой панели
    setLayout(new BorderLayout(15, 15));

    // Помещаем надпись с сообщением посередине окна.
    message = new MultiLineLabel(messageText, 20, 20, alignment);
    add(message, BorderLayout.CENTER);

    // Создаем панель для кнопок и размещаем ее внизу компонента
    // Panel. Указываем для нее менеджер компоновки FlowLayout.
    Panel buttonbox = new Panel();
    buttonbox.setLayout(new FlowLayout(FlowLayout.CENTER, 25, 15));
    add(buttonbox, BorderLayout.SOUTH);
```

```
    // Создаем все указанные кнопки, назначаем для каждой из них слушатель
    // событий и выполняемую команду и присоединяем их к панели buttonbox
    yes = new Button();          // Создаем кнопки
    no = new Button();
    cancel = new Button();
    // Добавляем кнопки в панель button box
    buttonbox.add(yes);
    buttonbox.add(no);
    buttonbox.add(cancel);

    // Регистрируем слушатели для каждой кнопки
    yes.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          fireEvent(new AnswerEvent(YesNoPanel.this,
                    AnswerEvent.YES));
        }
      });

    no.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          fireEvent(new AnswerEvent(YesNoPanel.this,
                    AnswerEvent.NO));
              }
      });
    cancel.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          fireEvent(new AnswerEvent(YesNoPanel.this,
                    AnswerEvent.CANCEL));
        }
      });

    // Вызываем методы установки свойств компонентов-сообщений
    // и компонентов-кнопок для помещения в них нужного текста
    setMessageText(messageText);
    setAlignment(alignment);
    setYesLabel(yesLabel);
    setNoLabel(noLabel);
    setCancelLabel(cancelLabel);
  }

  // Get-методы для всех свойств компонента.
  public String getMessageText() { return messageText; }
  public Alignment getAlignment() { return alignment; }
  public String getYesLabel() { return yesLabel; }
  public String getNoLabel() { return noLabel; }
  public String getCancelLabel() { return cancelLabel; }

  // Set-методы для всех свойств компонента.
  public void setMessageText(String messageText) {
    this.messageText = messageText;
    message.setLabel(messageText);
    validate();
  }
```

```
public void setAlignment(Alignment alignment) {
  this.alignment = alignment;
  message.setAlignment(alignment);
}

public void setYesLabel(String l) {
  yesLabel = l;
  yes.setLabel(l);
  yes.setVisible((l != null) && (l.length() > 0));
  validate();
}

public void setNoLabel(String l) {
  noLabel = l;
  no.setLabel(l);
  no.setVisible((l != null) && (l.length() > 0));
  validate();
}

public void setCancelLabel(String l) {
  cancelLabel = l;
  cancel.setLabel(l);
  cancel.setVisible((l != null) && (l.length() > 0));
  validate();
}

public void setFont(Font f) {
  super.setFont(f);  // Вызываем метод базового класса
  message.setFont(f);
  yes.setFont(f);
  no.setFont(f);
  cancel.setFont(f);
  validate();
}

/** В этом поле хранится список зарегистрированных слушателей событий. */
protected Vector listeners = new Vector();

/** Регистрируем слушатель действий, уведомляемый о нажатии кнопки */
public void addAnswerListener(AnswerListener l) {
  listeners.addElement(l);
}

/** Удаляем слушатель событий Answer из нашего списка
    заинтересованных слушателей */
public void removeAnswerListener(AnswerListener l) {
  listeners.removeElement(l);
}

/** Посылаем событие всем зарегистрированным слушателям */
public void fireEvent(AnswerEvent e) {
  // Делаем копию списка и, используя ее, активизируем события.
  // Это позволяет в ответ на текущее событие включать
  // в исходный список новые и удалять имеющиеся в нем слушатели.
  // Мы могли бы просто использовать перечислитель (enumerator) для этого
```

```
    // вектора, но при этом не создавалась бы настоящая копия списка.
    Vector list = (Vector) listeners.clone();
    for(int i = 0; i < list.size(); i++) {
      AnswerListener listener = (AnswerListener)list.elementAt(i);
      switch(e.getID()) {
      case AnswerEvent.YES: listener.yes(e); break;
      case AnswerEvent.NO: listener.no(e); break;
      case AnswerEvent.CANCEL: listener.cancel(e); break;
      }
    }
  }

  /** Метод main(), демонстрирующий класс */
  public static void main(String[] args) {
    // Создаем экземпляр InfoPanel с заданным заголовком и сообщением
    YesNoPanel p = new YesNoPanel("Вы действительно хотите закончить?");

    // Регистрируем для этой панели слушатель событий.
    // Он просто печатает результаты на консоль.
    p.addAnswerListener(new AnswerListener() {
        public void yes(AnswerEvent e) { System.exit(0); }
        public void no(AnswerEvent e) { System.out.println("No"); }
        public void cancel(AnswerEvent e) {
          System.out.println("Cancel");
        }
      });

    Frame f = new Frame();
    f.add(p);
    f.pack();
    f.setVisible(true);
  }
}
```

Пользовательские события

Компоненты могут использовать стандартные типы событий, определенные в пакетах `java.awt.event` **и** `javax.swing.event`**, но они не обязаны ограничиваться ими. Наш класс** `YesNoPanel` **определяет собственный тип событий –** `AnswerEvent`**. Определить новый класс событий на самом деле достаточно легко.** `AnswerEvent` **показан в примере 14.4.**

Пример 14.4. AnswerEvent.java

```
package com.davidflanagan.examples.beans;

/**
 * Класс YesNoPanel активизирует событие этого типа, когда
 * пользователь нажимает одну из его кнопок.
 * Поле id идентифицирует нажатую пользователем кнопку.
 **/
public class AnswerEvent extends java.util.EventObject {
```

```
  // Константы для кнопок
  public static final int YES = 0, NO = 1, CANCEL = 2;
  protected int id;                    // Какая кнопка была нажата?
  public AnswerEvent(Object source, int id) {
    super(source);
    this.id = id;
  }
  public int getID() { return id; }    // Возвращаем кнопку
}
```

Вместе с классом AnswerEvent в компоненте YesNoPanel определяется новый тип интерфейса слушателей событий, AnswerListener, содержащий методы, которые обязательно должны быть реализованы любым объектом, заинтересованным в получении уведомлений от компонента YesNoPanel. Определение интерфейса AnswerListener показано в примере 14.5.

Пример 14.5. AnswerListener.java

```
package com.davidflanagan.examples.beans;

/**
 * Классы, желающие получать уведомления о том, что пользователь
 * нажал кнопку компонента YesNoPanel, должны реализовать этот
 * интерфейс. Метод, который будет вызван, зависит от того,
 * какую кнопку нажал пользователь.
 **/
public interface AnswerListener extends java.util.EventListener {
  public void yes(AnswerEvent e);
  public void no(AnswerEvent e);
  public void cancel(AnswerEvent e);
}
```

Предоставление информации о компоненте

Сам класс YesNoPanel, как и классы MultiLineLabel, Alignment, AnswerEvent и AnswerListener, которыми он пользуется, является необходимой частью нашего компонента. Когда приложение, использующее данный компонент, распространяется, оно должно содержать все пять файлов класса. Однако часто имеются и другие виды классов, связанные с компонентом, но не предназначенные для использования разработчиками приложений. Эти классы используются контейнерным средством, которое манипулирует данным компонентом. Сам класс компонента не ссылается ни на какой из этих вспомогательных классов контейнера, поэтому он не зависит от них, и они не обязаны поставляться вместе с компонентом в составе законченного продукта.

Первым из этих необязательных, вспомогательных классов является класс BeanInfo. Как было объяснено выше, контейнер обнаруживает экспортируемые компонентом свойства, события и методы с помощью механизма интроспекции, основанного на программном интерфейсе

отражения Java. Разработчик компонента, желающий предоставить дополнительную информацию о компоненте или уточнить (несколько приблизительную) информацию, доступную через интроспекцию, должен для предоставления этой информации определить класс, реализующий интерфейс BeanInfo. Класс BeanInfo обычно наследуется от SimpleBeanInfo, который предоставляет пустую реализацию интерфейса BeanInfo. В том случае, когда вы хотите заместить только один или два метода, это проще сделать путем создания класса, производного от SimpleBeanInfo, чем реализовать BeanInfo непосредственно. Контейнерные инструментальные средства при поиске класса BeanInfo, относящегося к данному компоненту, предполагают использование следующего соглашения об именах: класс BeanInfo должен иметь то же имя, что и компонент, с добавлением в конец строки «BeanInfo». В примере 14.6 показана реализация класса YesNoPanelBeanInfo.

Данный класс BeanInfo определяет для нашего компонента несколько информационных частей:

- Изображение, представляющее компонент.
- Объект BeanDescriptor, содержащий ссылку на класс Customizer, относящийся к компоненту. Мы увидим реализацию этого класса ниже в данной главе.
- Список поддерживаемых свойств компонента с кратким описанием каждого. Некоторые контейнерные средства (но не *beanbox* фирмы Sun) показывают эти строки пользователю в некотором удобном для него виде.
- Метод, возвращающий наиболее часто используемое свойство компонента. Оно называется «свойством по умолчанию» (default property).
- Ссылка на класс PropertyEditor для одного из свойств. Мы увидим реализацию этого класса редактора свойств ниже в этой главе.

Помимо предоставления этой информации, класс BeanInfo может предоставлять информацию о методах, которые в нем определены, и событиях, которые он генерирует. Различные объекты FeatureDescriptor, предоставляющие информацию по свойствам и методам, могут содержать и другую информацию, не предоставляемую объектом YesNoPanelBeanInfo, такую как локализованное имя для отображения, которое отличается от программного имени.

Пример 14.6. YesNoPanelBeanInfo.java

```
package com.davidflanagan.examples.beans;
import java.beans.*;
import java.lang.reflect.*;
import java.awt.*;

/**
 * Этот класс BeanInfo предоставляет дополнительную информацию
 * о компоненте YesNoPanel помимо той, которая может быть
```

```
 * получена через механизм интроспекции.
 **/
public class YesNoPanelBeanInfo extends SimpleBeanInfo {
  /**
   * Возвращаем изображение для компонента. На самом деле
   * мы должны проверять значение аргумента kind, чтобы
   * выяснить размер изображения, требуемого контейнеру,
   * но поскольку мы можем предложить только одно изображение,
   * мы просто возвращаем его и предлагаем контейнеру работать с ним.
   **/
  public Image getIcon(int kind) { return loadImage("YesNoPanelIcon.gif"); }

  /**
   * Возвращаем дескриптор самого компонента. Он определяет
   * настройщик для класса компонента. Мы могли бы включить
   * сюда и строку описания
   **/
  public BeanDescriptor getBeanDescriptor() {
    return new BeanDescriptor(YesNoPanel.class, YesNoPanelCustomizer.class);
  }

  /** Это вспомогательный метод для создания объектов PropertyDescriptor */
  static PropertyDescriptor prop(String name, String description) {
    try {
      PropertyDescriptor p = new PropertyDescriptor(name, YesNoPanel.class);
      p.setShortDescription(description);
      return p;
    }
    catch(IntrospectionException e) { return null; }
  }

  // Инициализируем статический массив объектов PropertyDescriptor,
  // предоставляющий дополнительную информацию о поддерживаемых
  // компонентом свойствах. С помощью явного указания дескрипторов
  // для свойств мы можем предоставить для каждого свойства строку
  // с кратким описанием. Они не были бы доступны контейнеру при простом
  // использовании интроспекции. Кроме этого, мы можем для свойства
  // messageText зарегистрировать специальные редакторы свойств.
    static PropertyDescriptor[] props = {
    prop("messageText", "Текст сообщения, появляющийся в теле компонента"),
    prop("alignment", "Тип выравнивания текста сообщения"),
    prop("yesLabel", "Метка для кнопки Yes"),
    prop("noLabel", "Метка для кнопки No"),
    prop("cancelLabel","Метка для кнопки Cancel"),
    prop("font", "Шрифт для сообщения и кнопок"),
    prop("background", "Цвет фона"),
    prop("foreground", "Цвет переднего плана"),
  };
  static {
    props[0].setPropertyEditorClass(YesNoPanelMessageEditor.class);
  }

  /** Возвращаем дескрипторы свойств для данного компонента */
```

```
  public PropertyDescriptor[] getPropertyDescriptors() {
    return props;
  }

  /** Свойство message является наиболее часто используемым; делаем его
свойством по умолчанию */
  public int getDefaultPropertyIndex() { return 0; }
}
```

Создание простого редактора свойств

Компонент также может предоставлять используемые контейнерным средством вспомогательные классы `PropertyEditor`. `PropertyEditor` является достаточно гибким интерфейсом, предоставляющим компоненту возможность сообщить контейнеру, как нужно отображать и редактировать значения свойств определенных типов.

Для свойств обычных типов, таких как строки, числа, шрифты и цвета, контейнер всегда предоставляет простые редакторы свойств. Однако если у вашего компонента есть свойство нестандартного типа, вы должны зарегистрировать для этого типа свой редактор свойств. Самый простой способ зарегистрировать редактор свойств состоит в использовании простого соглашения об именах. Если ваш тип определен в классе *X*, редактор для него должен быть определен в классе *X*`Editor`. Другим возможным способом является регистрация редактора свойств путем вызова метода `PropertyEditorManager.registerEditor()`, например из конструктора вашего класса `BeanInfo`. В случае если вы будете вызывать этот метод из самого компонента, компонент будет зависеть от класса редактора свойств, поэтому редактор будет привязан к компоненту при использовании того в приложениях, что является нежелательным. Другой способ регистрации редактора свойства состоит в использовании в классе `BeanInfo` объекта `PropertyDescriptor` с целью указания `PropertyEditor` для отдельного свойства. Класс `YesNoPanelBeanInfo`, например, делает это для свойства `messageText`.

На первый взгляд интерфейс `PropertyEditor` может показаться запутанным. Его методы позволяют вам определить три способа отображения значения свойства и два способа редактирования значения свойства пользователем. Значение свойства может быть показано:

В виде строки

Если вы определите метод `getAsText()`, контейнер сможет преобразовать свойство в строку и отобразить ее пользователю.

В виде перечислимого значения

Если свойство может принимать только значения из некоторого фиксированного набора, вы можете определить метод `getTags()`, что позволит контейнеру отобразить всплывающее меню с разрешенными для этого свойства значениями.

В графической форме

Если вы определите метод `paintValue()`, контейнер может попросить редактор свойств отобразить значение, используя некоторый естественный графический формат, такой как цветовой образец для цветов. Вам также будет нужно определить метод `isPaintable()` для указания того, что графический формат поддерживается.

Двумя методами редактирования являются:

Строковое редактирование

Если вы определите метод `setAsText()`, контейнер будет знать, что он может просто предоставить пользователю возможность ввести значение в текстовое поле и передать его методу `setAsText()`. Если в вашем редакторе свойств определен метод `getTags()`, в нем также должен быть определен метод `setAsText()`, чтобы контейнер смог установить значение свойства, используя отдельные значения тегов.

Пользовательское редактирование

Если в вашем редакторе свойств определен метод `getCustomEditor()`, контейнер может вызвать его для получения некоторого графического компонента, который может быть отображен в диалоговом окне и служить в качестве пользовательского редактора для этого свойства. Вы также должны определить метод `supportsCustomEditor()`, чтобы указать, что пользовательское редактирование поддерживается.

Метод `setValue()` класса `PropertyEditor` вызывается для установки текущего значения свойства. Это то значение, которое методы `getAsText()` и `paintValue()` должны преобразовывать в строку или графическое представление.

Редактор свойств должен поддерживать список слушателей событий, которые интересуются изменениями значений свойства. Методы `addPropertyChangeListener()` и `removePropertyChangeListener()` являются стандартными методами регистрации и удаления слушателей событий. Когда редактор свойств изменяет значение свойства либо через `setAsText()`, либо посредством пользовательского редактора, он должен послать событие `PropertyChangeEvent` всем зарегистрированным слушателям.

В интерфейсе `PropertyEditor` определен метод `getJavaInitializationString()` для использования контейнерными инструментальными средствами, которые генерируют Java-код. Этот метод должен возвращать фрагмент Java-кода, который может инициализировать переменную, установив ее в текущее значение свойства.

И наконец, класс, реализующий интерфейс `PropertyEditor`, должен содержать конструктор без аргументов, чтобы контейнер мог динамически загружать его и создавать его экземпляры.

Большинство редакторов свойств могут быть гораздо проще, чем предлагается в нашем подробном описании. В большинстве случаев вы мо-

жете создавать классы, производные от PropertyEditorSupport, вместо непосредственной реализации интерфейса PropertyEditor. Этот полезный класс предоставляет пустые реализации для большинства методов PropertyEditor. Кроме этого он реализует методы для добавления и удаления слушателей событий.

Свойству, которое имеет перечислимые значения, необходим простой редактор свойств. Свойство alignment компонента YesNoPanel – достаточно типичный пример свойств этого типа. Свойство имеет только три допустимых значения, определенные в классе Alignment. Класс AlignmentEditor, показанный в примере 14.7, является редактором свойств, который сообщает контейнеру, как необходимо отображать и редактировать значение этого свойства. Поскольку AlignmentEditor следует соглашению JavaBeans об именах, контейнер автоматически использует его для всех свойств с типом Alignment.

Пример 14.7. AlignmentEditor.java

```
package com.davidflanagan.examples.beans;
import java.beans.*;
import java.awt.*;

/**
 * Этот PropertyEditor определяет перечислимые значения для
 * свойства alignment, чтобы контейнер или среда разработки
 * могли предоставить эти значения на выбор пользователю
 **/
public class AlignmentEditor extends PropertyEditorSupport {
  /** Возвращаем список имен значений для перечислимого типа */
  public String[] getTags() {
    return new String[] { "left", "center", "right" };
  }

  /** Преобразуем каждое из имен этих значений в реальное значение. */
  public void setAsText(String s) {
    if (s.equals("left")) setValue(Alignment.LEFT);
    else if (s.equals("center")) setValue(Alignment.CENTER);
    else if (s.equals("right")) setValue(Alignment.RIGHT);
    else throw new IllegalArgumentException(s);
  }

  /** Это важный метод для генерации кода */
  public String getJavaInitializationString() {
    Object o = getValue();
    if (o == Alignment.LEFT)
      return "com.davidflanagan.examples.beans.Alignment.LEFT";
    if (o == Alignment.CENTER)
      return "com.davidflanagan.examples.beans.Alignment.CENTER";
    if (o == Alignment.RIGHT)
      return "com.davidflanagan.examples.beans.Alignment.RIGHT";
    return null;
  }
}
```

Создание сложного редактора свойств

У компонента YesNoPanel есть еще одно свойство, которому требуется редактор свойств. Свойство messageText может содержать многострочное сообщение, которое будет отображаться на панели. Этому свойству необходим редактор свойств из-за того, что контейнерная программа не отличает однострочные строковые типы от многострочных. Объекты TextField, которые она использует для ввода текста, не позволяют пользователю вводить несколько строк текста. По этой причине вы должны определить класс YesNoPanelMessageEditor и зарегистрировать его с помощью PropertyDescriptor для свойства message, как показано в примере 14.6.

В примере 14.8 показано определение этого редактора свойств. Он представляет собой более сложный редактор, поддерживающий создание пользовательского компонента-редактора и графической формы представления значения. Обратите внимание, что в этом примере реализуется непосредственно PropertyEditor. Это означает, что он должен обрабатывать регистрацию и рассылку уведомлений объектам PropertyChangeListener. Метод getCustomEditor() возвращает компонент-редактор для многострочного текста. На рис. 14.2 показан этот пользовательский редактор, помещенный в диалоговое окно, созданное контейнерной программой. Обратите внимание, что кнопка **Done** на этом рисунке является частью диалогового окна контейнера, а не частью самого редактора свойств.

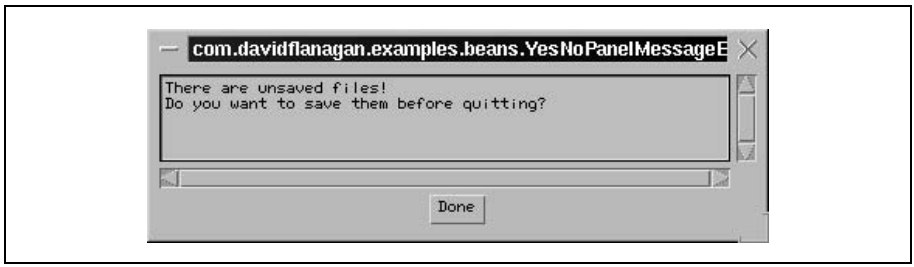


***Рис. 14.2.** Окно редактора пользовательского свойства*

Метод paintValue() отображает значение свойства messageText. Это многострочное значение обычно не помещается в небольшом прямоугольном участке экрана, отведенном для этого свойства, поэтому paintValue() выводит инструкции для вызова пользовательского редактора, позволяющего пользователю видеть и редактировать значение свойства.

Пример 14.8. YesNoPanelMessageEditor.java

```
package com.davidflanagan.examples.beans;
import java.beans.*;
import java.awt.*;
```

```
import java.awt.event.*;

/**
 * Этот класс является пользовательским редактором свойства
 * messageText компонента YesNoPanel. Он необходим, потому
 * что редактор, используемый по умолчанию для свойств типа
 * String, не позволяет вводить многострочный текст.
*/
public class YesNoPanelMessageEditor implements PropertyEditor {
  protected String value; // Значение, которое мы будем редактировать

  public void setValue(Object o) { value = (String) o; }
  public Object getValue() { return value; }
  public void setAsText(String s) { value = s; }
  public String getAsText() { return value; }
  public String[] getTags() { return null; } // неперечислимое; без тегов

  // Сообщаем, что мы разрешаем пользовательское редактирование
  public boolean supportsCustomEditor() { return true; }

  // Возвращаем пользовательский редактор. Просто создаем
  // и возвращаем объект TextArea для редактирования
  // многострочного текста. А также регистрируем слушатель событий
  // от этой текстовой области, чтобы обновлять значения после
  // пользовательского ввода и активизировать события изменения свойств,
  // которые редакторы свойств должны генерировать.
  public Component getCustomEditor() {
    final TextArea t = new TextArea(value);
    // У компонента TextArea нет размеров по умолчанию,
    // поэтому устанавливаем их
    t.setSize(300, 150);
    t.addTextListener(new TextListener() {
        public void textValueChanged(TextEvent e) {
          value = t.getText();
          listeners.firePropertyChange(null, null, null);
        }
      });
    return t;
  }

  // Графическое представление значения для использования
  // в пользовательском редакторе. Просто выводим некое изображение
  // и надеемся, что оно уместится в отведенном для него прямоугольнике.
  // Можно было бы сделать и посложнее.
  public boolean isPaintable() { return true; }
  public void paintValue(Graphics g, Rectangle r) {
    g.setClip(r);
    g.drawString("Click to edit...", r.x+5, r.y+15);
  }

  // Важный метод для генерации кода. Обратите внимание, что перед
  // возвращением строки необходимо предварять кавычки и обратные слэши
  // символом "\".
public String getJavaInitializationString() { return "\"" + value + "\""; }
```

```
    // В этом коде используется класс PropertyChangeSupport для хранения
    // списка слушателей, заинтересованных в изменениях, которые мы вносим
    // в значения.
  protected PropertyChangeSupport listeners =new PropertyChangeSupport(this);
    public void addPropertyChangeListener(PropertyChangeListener l) {
      listeners.addPropertyChangeListener(l);
    }
    public void removePropertyChangeListener(PropertyChangeListener l) {
      listeners.removePropertyChangeListener(l);
    }
  }
```

Создание настройщика компонентов

Компонент может пожелать предоставить пользователю контейнерной программы некоторый способ настройки его свойств помимо установки их по одному во время выполнения. Компонент может сделать это, создав для себя класс `Customizer` (настройщик) и зарегистрировав его при помощи объекта `BeanDescriptor`, возвращаемого его классом `BeanInfo`, как показано в примере 14.6.

Настройщик должен быть графическим компонентом некоторого вида, подходящего для отображения в диалоговом окне, созданном контейнером. Поэтому класс настройщика обычно является производным от класса `Panel`. Кроме этого настройщик должен реализовывать интерфейс `Customizer`. Этот интерфейс содержит методы, используемые для добавления и удаления слушателей событий, возникающих при изменении свойства, и метод `setObject()`, вызываемый контейнером для того, чтобы сообщить настройщику, какой объект он настраивает. Всякий раз, когда пользователь посредством настройщика делает изменения в компоненте, настройщик посылает событие `PropertyChangeEvent` всем заинтересованным в нем слушателям. И наконец, как и у редактора свойств, в состав настройщика должен входить конструктор без аргументов, чтобы контейнер мог легко создать экземпляр объекта его типа.

В примере 14.9 показан настройщик для нашего компонента `YesNoPanel`. Этот настройщик выводит на экран панель, имеющую такое же расположение, как и `YesNoPanel`, но он заменяет объект `TextArea` на окно сообщений и три объекта `TextField`, соответствующие трем кнопкам, которые отображаются в диалоговом окне. В эти текстовые области пользователь может вводить значения свойств `messageText`, `yesLabel`, `noLabel` и `cancelLabel`. На рис. 14.3 показана панель настройщика, помещенная в диалоговое окно, созданное контейнерной программой. И снова обратите внимание, что кнопка **Done** является частью диалогового окна контейнера, а не частью самого настройщика.

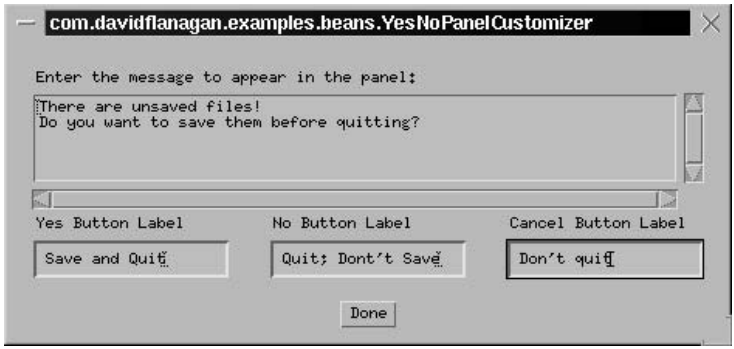


Рис. 14.3. *Диалоговое окно настройщика для компонента YesNoPanel*

Пример 14.9. YesNoPanelCustomizer.java

```
package com.davidflanagan.examples.beans;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;

/**
 * Этот класс является настройщиком компонента YesNoPanel.
 * Он выводит на экран область TextArea и три поля TextFields,
 * в которые пользователь может вводить основное сообщение и надписи
 * для каждой из трех кнопок. Он не позволяет задавать свойство alignment.
  **/
public class YesNoPanelCustomizer extends Panel
  implements Customizer, TextListener
{
  protected YesNoPanel bean;   // Настраиваемый компонент
  protected TextArea message;  // Для ввода сообщения
  protected TextField fields[];// Для ввода текста на кнопках

  // Контейнер вызывает этот метод, чтобы сообщить нам, какой компонент будет
  // настраиваться. Данный метод всегда вызывается перед тем, как настройщик
  // будет показан на экране, поэтому здесь можно безопасно создавать
  // для него графический интерфейс.
  public void setObject(Object o) {
    bean = (YesNoPanel)o;  // Сохраняем настраиваемый объект

    // Помещаем надпись наверху панели.
    this.setLayout(new BorderLayout());
    this.add(new Label("Enter the message to appear in the panel:"),
         "North");
    // Перевод: "Введите сообщение для вывода на панель:"

    // Размещаем под ней большую текстовую область
    // для ввода сообщения.
    message = new TextArea(bean.getMessageText());
```

```
    message.addTextListener(this);
    // Компонент TextArea не знает, какие размеры желательны.
    // Вы должны сообщить ему.
    message.setSize(400, 200);
    this.add(message, "Center");

    // Теперь добавляем ряд текстовых полей для ввода
    // надписей на кнопках.
    Panel buttonbox = new Panel();  // Контейнер для ряда
    // Равные интервалы
    buttonbox.setLayout(new GridLayout(1, 0, 25, 10));
    this.add(buttonbox, "South");   // Размещаем ряд внизу

    // Теперь собираемся создать три объекта TextField,
    // чтобы разместить их в этом ряду. Но нам нужно над каждым
    // из них поместить надпись, поэтому для каждой комбинации
    // TextField+Label создаем свой контейнер.
    fields = new TextField[3];       // Массив полей TextField.
    String[] labels = new String[] { // Надписи для каждого поля.
      "Yes Button Label", "No Button Label", "Cancel Button Label"};
    // Перевод: "Текст кнопки Yes", "Текст кнопки No", "Текст кнопки Cancel"
    String[] values = new String[] { // Начальные значения для каждого.
      bean.getYesLabel(), bean.getNoLabel(), bean.getCancelLabel()};
    for(int i = 0; i < 3; i++) {
      Panel p = new Panel();         // Создаем контейнер.
      // Устанавливаем для него BorderLayout.
      p.setLayout(new BorderLayout());
      // Помещаем вверху надпись.
      p.add(new Label(labels[i]), "North");
      // Создаем текстовое поле.
      fields[i] = new TextField(values[i]);
      // Размещаем его под надписью.
      p.add(fields[i], "Center");
      // Устанавливаем слушатель событий.
      fields[i].addTextListener(this);
      buttonbox.add(p);    // Добавляем контейнер в строку.
    }
  }
  // Добавляем немного пустого пространства вокруг панели.
  public Insets getInsets() { return new Insets(10, 10, 10, 10); }

  // Этот метод определен в интерфейсе TextListener. Он будет вызываться
  // всякий раз, когда пользователь вводит новый символ в текстовую область
  // или в текстовое поле. Он обновляет соответствующее свойство компонента
  // и активизирует событие по его изменению, что требуется от всех
  // настройщиков. Заметим, что от нас не требуется активизировать событие
  // при каждом нажатии клавиши. Вместо этого мы могли бы ввести кнопку
  // "Применить", чтобы делать все изменения за один раз в одном событии
  // изменения свойства.
  public void textValueChanged(TextEvent e) {
    TextComponent t = (TextComponent)e.getSource();
    String s = t.getText();
```

```
    if (t == message) bean.setMessageText(s);
    else if (t == fields[0]) bean.setYesLabel(s);
    else if (t == fields[1]) bean.setNoLabel(s);
    else if (t == fields[2]) bean.setCancelLabel(s);
    listeners.firePropertyChange(null, null, null);
  }

  // В этом коде используется класс PropertyChangeSupport для хранения списка
  // слушателей, заинтересованных в изменениях, которые мы вносим в значения.
  protected PropertyChangeSupport listeners =new
PropertyChangeSupport(this);
  public void addPropertyChangeListener(PropertyChangeListener l) {
    listeners.addPropertyChangeListener(l);
  }
  public void removePropertyChangeListener(PropertyChangeListener l) {
    listeners.removePropertyChangeListener(l);
  }
}
```

Упражнения

14-1. В главах 10, 12 и 13 содержатся примеры AWT- и Swing-компонентов, позволяющие пользователю рисовать с помощью мыши. Выберите один из этих классов, переименуйте его в `ScribbleBean` и упакуйте в файл JAR (вместе с другими классами примера, которые ему могут потребоваться). Теперь установите его в любое контейнерное приложение по вашему выбору, чтобы убедиться, что он работает. Введите в ваш компонент метод `erase()`, который очищает рисунок, и создайте с помощью контейнера кнопку, вызывающую этот метод.

14-2. Измените ваш компонент `ScribbleBean` так, чтобы он содержал свойства `color` и `width`, задающие цвет и толщину линий, используемых для рисования. Еще раз упакуйте компонент и проверьте свойства в контейнере.

14-3. Приложению, использующему компонент `ScribbleBean`, может понадобиться получать уведомления о том, что пользователь завершил очередной штрих (stroke) рисунка (то есть каждый раз, когда пользователь нажал кнопку мыши, передвинул ее, а затем отпустил). Например, приложение могло бы делать внеэкранную копию рисунка после завершения каждого штриха, чтобы реализовать возможность отмены (undo) сделанных изменений. Чтобы предоставить такой вид уведомлений, измените ваш компонент `ScribbleBean`, введя в него поддержку события «stroke». Создайте простой класс `StrokeEvent` и интерфейс `StrokeListener`. Измените компонент `ScribbleBean` так, чтобы он позволял регистрировать и удалять объекты `StrokeListener` и рассылал всем зарегистрированным слушателям уведомления по завершении рисо-

вания каждого штриха рисунка. Снова сгенерируйте файл JAR для вашего компонента так, чтобы он включал в себя файлы классов `StrokeEvent` и `StrokeListener`.

14-4. Создайте для компонента `ScribbleBean` подкласс `BeanInfo`. Данный класс должен предоставлять информацию об определенных в компоненте методе `erase()`, свойствах `color` и `width` и событии `stroke`. Класс `BeanInfo` должен использовать метод `FeatureDescriptor.setShortDescription()`, чтобы предоставлять простые описательные строки для самого компонента, а также его метода, свойств и события.

Глава 15

Апплеты

Эта глава демонстрирует технику написания апплетов. Она начнется с простейшего апплета «Hello World» («Здравствуй, мир») и закончится более сложными апплетами. Вы узнаете, как:

- создавать изображения в вашем апплете;
- обрабатывать и отвечать на простую информацию, вводимую пользователем;
- считывать и использовать параметры апплета для его настройки;
- загружать и отображать изображения, а также загружать и проигрывать звуковые фрагменты;
- упаковывать апплет и относящиеся к нему файлы в файл JAR.

Знакомство с апплетами

Апплет (applet), как видно из его названия, является мини-приложением, созданным для запуска веб-броузером или в контексте некоторого другого «визуализатора апплетов» (applet viewer). У апплетов есть несколько отличий от обычных приложений. Одно из самых важных – ряд ограничений по безопасности, определяющих набор разрешенных операций. Апплеты достаточно часто содержат ненадежный код, из-за чего им нельзя разрешать, например, доступ к файловой системе.

Все апплеты являются подклассами `java.applet.Applet`, который, в свою очередь, наследует от `java.awt.Panel` и `java.awt.Component`. Поэтому создание апплетов больше похоже на написание графического компонента, чем приложения. В частности, у апплетов нет ни метода `main()`, ни какой-либо другой отдельной точки входа, с которой бы эта программа начинала выполнение. Вместо этого для написания апплета необходи-

мо создать подкласс класса Applet и заместить (override) несколько стандартных методов. В нужное время, при возникновении определенных ситуаций, броузер или визуализатор апплетов вызывает эти определенные вами методы. Апплет не может управлять вызвавшим его потоком исполнения, он просто отсылает ответ броузеру или визуализатору апплетов, когда он его об этом попросит. По этой причине созданные вами методы должны немедленно выполнять необходимые действия и возвращать управление. Им не разрешается входить в длительные (или бесконечные) циклы. Для выполнения длительных или циклически повторяющихся действий, таких как воспроизведение анимации, апплет должен создать собственный поток исполнения, которым он полностью управляет.

Задача написания апплетов сводится, таким образом, к определению соответствующих методов. Некоторые из этих методов определены в классе Applet:

`Init()`

Вызывается при первой загрузке апплета в броузер или визуализатор. Действия, связанные с инициализацией апплета, обычно выполняются в нем, а не в методе-конструкторе. (Броузер не передает никаких аргументов в метод-конструктор апплета, поэтому определять последний не слишком полезно.)

`Destroy()`

Вызывается перед выгрузкой апплета из броузера или визуализатора. Он должен освободить все ресурсы, кроме памяти, которые захватил апплет.

`Start()`

Вызывается, когда апплет становится видимым и должен начать выполнять то, ради чего он создан. Часто используется при анимации и управлении потоками.

`Stop()`

Вызывается, когда апплет временно становится невидимым, например, если пользователь прокрутил окно, удалив апплет с экрана. Сообщает апплету, что нужно прекратить выполнение анимации или других задач.

`GetAppletInfo()`

Вызывается для получения информации об апплете. Должен возвращать строку, пригодную для отображения в диалоговом окне.

`GetParameterInfo()`

Вызывается для получения информации о параметрах, обрабатываемых апплетом. Должен возвращать строку, описывающую эти параметры.

Кроме этих методов класса Applet, есть также ряд методов, унаследованных от базовых классов класса Applet, которые вызываются броузе-

ром в определенные моменты времени и которые должны быть замещены апплетом. Наиболее очевидный из них – метод `paint()`, вызываемый броузером или визуализатором для указания апплету отобразить себя на экране. Близко связан с ним метод `print()`, который апплет должен заместить в случае, если он хочет отображать себя на бумаге не так, как на экране. Есть еще несколько методов, которые должны быть замещены апплетом для реализации откликов на события. Например, если апплету необходимо отвечать на нажатия кнопок мыши, он должен заместить метод `mouseDown()`. (Мы подробнее поговорим об обработке событий далее в этой главе.)

Кроме того, в классе `Applet` определены несколько методов, обычно используемых (но не замещаемых) апплетом:

`GetImage()`

Загружает из сети файл с изображением и возвращает объект `java.awt.Image`.

`GetAudioClip()`

Загружает из сети звуковой клип и возвращает объект `java.applet.AudioClip`.

`GetParameter()`

Ищет и возвращает значение именованного параметра, указанного в файле HTML в соответствующем этому апплету теге `<PARAM>`.

`GetCodeBase()`

Возвращает базовый URL, с которого был загружен файл с классом данного апплета.

`GetDocumentBase()`

Возвращает базовый URL HTML-файла, который ссылается на данный апплет.

`ShowStatus()`

Отображает сообщение в строке состояния броузера или визуализатора апплетов.

`GetAppletContext()`

Возвращает объект `java.applet.AppletContext`, соответствующий данному апплету. `AppletContext` определяет полезный метод `showDocument()`, который просит броузер загрузить и отобразить новую вебстраницу.

Первый апплет

В примере 15.1 приведен, возможно, самый простой апплет, который вы можете написать на Java. На рис. 15.1 показан генерируемый им вывод. Этот пример представляет метод `paint()`, вызываемый визуали-

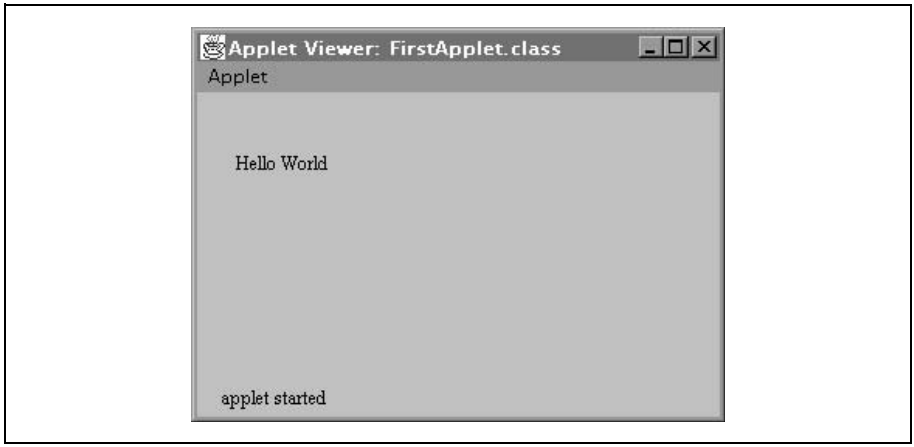


Рис. 15.1. *Простой апплет*

затором (или броузером) при необходимости отобразить апплет. Этот метод должен реализовать для вашего апплета графический вывод, такой как рисование текста или линий или отображение изображений. Аргументом метода `paint()` является объект `java.awt.Graphics`, используемый вами для рисования.

Пример 15.1. FirstApplet.java

```
package com.davidflanagan.examples.applet;
import java.applet.*;  // Не забудьте этот важный оператор!
import java.awt.*;     // Или этот, необходимый для графики!

/** Этот апплет просто говорит "Hello World" ("Здравствуй, мир!") */
public class FirstApplet extends Applet {
  // Этот метод отображает апплет.
  public void paint(Graphics g) {
    g.drawString("Здравствуй, мир!", 25, 50);
  }
}
```

Для того чтобы отобразить апплет, нам нужен ссылающийся на него HTML-файл. Ниже приведен фрагмент HTML-файла, который можно использовать с нашим первым апплетом:

```
<applet code="com.davidflanagan.examples.applet.FirstApplet.class"
codebase="../../../../"
width=150 height=100>
</applet>
```

Теперь с помощью HTML-файла, содержащего ссылку на апплет, вы можете просмотреть апплет в визуализаторе апплетов или броузере. Обратите внимание на то, что атрибуты `WIDTH` и `HEIGHT` в этом HTML-теге являются обязательными. Для большинства апплетов, приведенных в примерах данной книги, я привожу только Java-код без соответ-

ствующего ему HTML-файла. Чаще всего HTML-файл содержит такой же простой тег, как и приведенный здесь.

Апплет Clock

Clock представляет собой апплет, отображающий текущее время, как показано на рис. 15.2, и обновляющий его один раз в секунду. В отличие от апплета из примера 15.1, который определяет метод paint() и самостоятельно выполняет вывод текста при помощи метода Graphics.drawString(), в данном примере для вывода используется компонент java.awt.Label. Хотя апплеты достаточно часто используют для рисования метод paint(), необходимо помнить, что апплеты расширяют интерфейс java.awt.Panel и поэтому могут сами содержать любые типы графических компонентов. В апплете Clock определен метод init(), создающий и конфигурирующий компонент Label.

***Рис. 15.2.** Апплет Clock*

Для того чтобы обновлять время каждую секунду, апплет Clock реализует интерфейс Runnable и создает объект Thread, запускающий метод run(). Методы апплета start() и stop() вызываются броузером, когда апплет становится видимым или скрытым. Они запускают и останавливают поток исполнения. (Хотя пример написан для использования с Java 1.1, он не использует метод Thread.stop(), не рекомендуемый в Java 1.2 .)

И наконец, для предоставления информации о себе апплет Clock реализует метод getAppletInfo(). Некоторые из визуализаторов апплетов предоставляют интерфейс для вывода этой информации.

Пример 15.2. Clock.java

```
package com.davidflanagan.examples.applet;
import java.applet.*;         // Не забудьте этот важный оператор!
import java.awt.*;            // И этот, необходимый для графики!
import java.util.Date;        // Для получения текущего времени
import java.text.DateFormat; // Для отображения времени

/**
 * Этот апплет отображает время и обновляет его каждую секунду
 **/
public class Clock extends Applet implements Runnable {
  Label time;                 // Компонент для отображения времени
```

```
DateFormat timeFormat;    // Этот объект преобразует время в строку
Thread timer;             // Поток, обновляющий время
volatile boolean running; // Флаг, используемый для остановки потока

/**
 * Метод init() вызывается при первом запуске апплета броузером.
 * Он настраивает компонент Label и получает объект DateFormat
 **/
public void init() {
  time = new Label();
  time.setFont(new Font("helvetica", Font.BOLD, 12));
  time.setAlignment(Label.CENTER);
  setLayout(new BorderLayout());
  add(time, BorderLayout.CENTER);
  timeFormat = DateFormat.getTimeInstance(DateFormat.MEDIUM);
}

/**
 * Броузер вызывает этот метод для запуска апплета. Здесь мы создаем
 * и запускаем поток, который будет обновлять время каждую секунду.
 * Заметим, что мы стараемся никогда не иметь больше одного потока
 **/
public void start() {
  running = true;             // Устанавливаем флаг
  if (timer == null) {        // Если у нас еще нет потока,
    timer = new Thread(this); // то создаем
    timer.start();            // и запускаем его
  }
}

/**
 * Этот метод реализует интерфейс Runnable. Это тело потока. Раз в секунду
 * он обновляет текст надписи для отображения текущего времени
 **/
public void run() {
  while(running) {   // Цикл, пока нас не остановят
    // Получаем текущее время, преобразуем его в строку
    // и отображаем в компоненте Label
    time.setText(timeFormat.format(new Date()));
    // Ждем 1000 миллисекунд
    try { Thread.sleep(1000); }
    catch (InterruptedException e) {}
  }
  // Если поток завершается, устанавливаем его в null, чтобы при следующем
  // вызове метода start() мы смогли создать новый.
  timer = null;
}

/**
 * Броузер вызывает этот метод, чтобы сообщить апплету,
 * что он стал невидимым и не должен выполняться.
 * Он устанавливает флаг, предписывающий методу run() завершиться
 **/
```

```
    public void stop() { running = false; }

    /**
     * Возвращаем информацию об апплете для отображения в визуализаторе
     **/
    public String getAppletInfo() {
      return "Clock applet Copyright (c) 2000 by David Flanagan";
    }
}
```

Апплеты и модель событий Java 1.0

При переходе от Java 1.0 к Java 1.1 была значительно изменена модель событий AWT. В главе 10 описывалась исключительно обработка событий Java 1.1, так как модель событий Java 1.0 объявлена устаревшей. Однако из-за большого числа используемых броузеров (таких как Netscape 3.0 и 4.0), поддерживающих только модель событий Java 1.0, апплеты все еще иногда создаются с использованием этой модели. В этом разделе вкратце описывается обработка событий Java 1.0 и приводится пример использующего ее апплета.[1]

В Java 1.0 все события представлены классом `java.awt.Event`. У этого класса есть несколько полей экземпляра, описывающих эти события. В одном из этих полей, `id`, указывается тип события. В классе `Event` определен ряд констант, являющихся возможными значениями поля `id`. Поле `target` указывает объект (как правило, `Component`), сгенерировавший данное событие или в котором событие возникло (то есть источник события). Другие поля могут использоваться или нет, в зависимости от типа события. Например, поля `x` и `y` определены, когда поле `id` имеет значение `BUTTON_EVENT`, и не определены в случае `ACTION_EVENT`. Поле `arg` может предоставлять дополнительную информацию, специфичную для конкретного типа события.

События Java 1.0, прежде всего, перенаправляются методу `handleEvent()` компонента, в котором они возникают. Реализация этого метода по умолчанию анализирует поле `id` объекта `Event` и пересылает наиболее часто используемые типы событий различным зависящим от события методам, перечисленным в табл. 15.1.

Таблица 15.1. Методы обработки событий компонента в Java 1.0

`action()`	`lostFocus()`	`mouseExit()`
`gotFocus()`	`mouseDown()`	`mouseMove()`
`keyDown()`	`mouseDrag()`	`mouseUp()`
`keyUp()`	`mouseEnter()`	

[1] Данный раздел и пример из него являются выдержкой из книги «Java Foundation Classes in a Nutshell».

Перечисленные в табл. 15.1 методы определены в классе Component. Одна из основных характеристик модели событий Java 1.0 состоит в том, что для обработки событий вы должны заместить эти методы в своем классе. То есть чтобы определить пользовательский механизм обработки событий, вы должны создать производный класс таким же образом, как вы делаете это, например, при написании апплетов. Таким образом, если вы заинтересованы в получении события LIST_SELECT или WINDOW_ICONIFY, вам потребуется заместить непосредственно метод handleEvent(), а не один из более конкретных методов. Делая так, вы, как правило, должны вызывать метод super.handleEvent() для используемой по умолчанию диспетчеризации остальных событий.

Метод handleEvent(), как и все остальные методы для событий определенного типа, возвращает результат типа boolean. Если метод обработки событий возвращает значение false, как все они делают по умолчанию, это означает, что событие не было обработано, следовательно, оно должно быть передано контейнеру текущего компонента, чтобы выяснить, не желает ли он его обработать. В противном случае, если метод возвращает значение true, он сигнализирует этим, что событие было обработано и не требует никакой дальнейшей обработки.

Тот факт, что необработанные события передаются вверх по иерархии контейнеров, чрезвычайно важен. Это означает, что вы можете заместить, например, метод action() апплета, чтобы обрабатывать события ACTION_EVENT, генерируемые кнопками этого апплета. Если бы они так не распространялись, вам пришлось бы создавать пользовательский подкласс класса Button для каждой кнопки, которую вы хотите добавить к интерфейсу.

В модели Java 1.0 нет стандартного способа узнать, какие типы событий генерируются каждым типом графического компонента и какие поля объекта Event заполняются конкретным типом события. Вы должны просто поискать эту информацию в документации по определенному AWT-компоненту.

Многие типы событий используют поле modifiers объекта Event для указания, какие клавиши-модификаторы были нажаты при возникновении события. Это поле содержит битовую маску определенных в классе Event констант SHIFT_MASK, CTRL_MASK, META_MASK и ALT_MASK. Для проверки различных клавиш-модификаторов могут использоваться методы shiftDown(), controlDown() и metaDown(). В случае возникновения события от мыши в классе Event не предусмотрено никакое специальное поле для указания того, какая кнопка мыши была нажата. Вместо этого данную информацию можно извлечь с помощью констант клавиш-модификаторов. Это дает возможность таким системам, как Macintosh, использующим однокнопочную мышь, эмулировать другие кнопки с помощью клавиш-модификаторов. Если используется левая кнопка мыши, не указывается ни один модификатор. Если используется правая кнопка, в поле modifier устанавливается бит META_MASK, а если нажата средняя кнопка, – бит ALT_MASK.

При возникновении событий от клавиатуры вы должны проверять поле `id` объекта `Event`, чтобы выяснить, клавиша какого типа была нажата. Если тип события – `KEY_PRESS` или `KEY_RELEASE`, нажатая клавиша имеет ASCII- или Unicode-представление, а поле `key` объекта события содержит код данной клавиши. С другой стороны, если `id` имеет значение `KEY_ACTION` или `KEY_ACTION_RELEASE`, была нажата функциональная клавиша некоторого типа, а поле `field` содержит одну из клавиатурных констант, определенных в классе `Event`, такую как `Event.F1` или `Event.LEFT`.

Держите в уме это краткое введение в модель событий Java 1.0, когда будете изучать пример 15.3. Этот апплет предоставляет пользователю возможность рисовать с помощью мыши. Также он позволяет стирать его рисунок, щелкнув на кнопке или нажав клавишу <E>. Как вы увидите, для обработки событий от мыши и клавиатуры и события, генерируемого при нажатии кнопки, апплет замещает методы базового класса. В частности, обратите внимание, что методы обработки событий возвращают значение типа `boolean`. В этом апплете не определен метод `paint()`. Для простоты он выполняет рисование непосредственно в ответ на принимаемые им события и не запоминает их координат. Это значит, что он не сможет снова воспроизвести рисунок пользователя, если он будет на какое-то время выведен за пределы экрана в результате прокрутки.

Пример 15.3. Scribble.java

```
package com.davidflanagan.examples.applet;
import java.applet.*;
import java.awt.*;

/**
 * Этот апплет позволяет пользователю рисовать при помощи мыши.
 * Он демонстрирует модель событий Java 1.0
 **/
public class Scribble extends Applet {
  private int lastx, lasty;   // Запоминаем последнюю координату мыши.
  Button erase_button;        // Кнопка Erase.

  /** Инициализируем кнопку Erase, запрашиваем фокус ввода */
  public void init() {
    erase_button = new Button("Erase");
    this.add(erase_button);
    // Устанавливаем цвет фона рисунка
    this.setBackground(Color.white);
    // Запрашиваем фокус ввода и получаем события клавиатуры
    this.requestFocus();
  }

  /** Отвечаем на нажатие кнопки мыши */
  public boolean mouseDown(Event e, int x, int y) {
    lastx = x; lasty = y;   // Запоминаем, где было нажатие
    return true;
  }
```

```
  /** Отвечаем на перемещение мыши */
  public boolean mouseDrag(Event e, int x, int y) {
    Graphics g = getGraphics();
    // Проводим линию от последней позиции до этой
    g.drawLine(lastx, lasty, x, y);
    // И запоминаем новую последнюю позицию
    lastx = x; lasty = y;
    return true;
  }

  /** Отвечаем на нажатие клавиши: Удаляем рисунок,
      когда пользователь вводит 'e' */
  public boolean keyDown(Event e, int key) {
    if ((e.id == Event.KEY_PRESS) && (key == 'e')) {
      Graphics g = getGraphics();
      g.setColor(this.getBackground());
      g.fillRect(0, 0, bounds().width, bounds().height);
      return true;
    }
    else return false;
  }

  /** Отвечаем на нажатия кнопки: удаляем рисунок,
      когда пользователь нажимает кнопку */
  public boolean action(Event e, Object arg) {
    if (e.target == erase_button) {
      Graphics g = getGraphics();
      g.setColor(this.getBackground());
      g.fillRect(0, 0, bounds().width, bounds().height);
      return true;
    }
    else return false;
  }
}
```

Подробности о событиях Java 1.0

В примере 15.4 показан апплет, обрабатывающий все входные события от пользователя, которые могут произойти в апплете, и выводящий подробную информацию о них. Это в основном события от клавиатуры и мыши. В программе не создаются никакие компоненты пользовательского интерфейса, поэтому она не обрабатывает высокоуровневые сообщения, которые эти компоненты могли бы генерировать. Данный пример интересен тем, что в нем иллюстрируется, как нужно интерпретировать клавиши-модификаторы и как использовать различные типы клавиатурных событий. Если вам придется писать код со сложной обработкой ошибок, вы можете смоделировать его фрагменты так же, как это сделано в данном примере.

Пример 15.4. EventTester.java

```
package com.davidflanagan.examples.applet;
import java.applet.*;
```

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

/** Апплет, дающий подробную информацию о событиях Java 1.0 */
public class EventTester extends Applet {
  // Обрабатываем события от мыши
  public boolean mouseDown(Event e, int x, int y) {
    showLine(mods(e.modifiers) + "Mouse Down: [" + x + "," + y + "]");
    return true;
  }
  public boolean mouseUp(Event e, int x, int y) {
    showLine(mods(e.modifiers) + "Mouse Up: [" + x + "," + y + "]");
    return true;
  }
  public boolean mouseDrag(Event e, int x, int y) {
    showLine(mods(e.modifiers) + "Mouse Drag: [" + x + "," + y + "]");
    return true;
  }
  public boolean mouseMove(Event e, int x, int y) {
    showLine(mods(e.modifiers) + "Mouse Move: [" + x + "," + y + "]");
    return true;
  }
  public boolean mouseEnter(Event e, int x, int y) {
    showLine("Mouse Enter: [" + x + "," + y + "]"); return true;
  }
  public boolean mouseExit(Event e, int x, int y) {
    showLine("Mouse Exit: [" + x + "," + y + "]"); return true;
  }

  // Обрабатываем события смены фокуса
  public boolean gotFocus(Event e, Object what) {
    showLine("Got Focus"); return true;
  }
  public boolean lostFocus(Event e, Object what) {
    showLine("Lost Focus"); return true;
  }

  // Обрабатываем события нажатий и отпусканий клавиш
  // Это несколько запутанно, т. к. есть два типа событий от клавиатуры
  public boolean keyDown(Event e, int key) {
    int flags = e.modifiers;
    if (e.id == Event.KEY_PRESS)                // обычная клавиша
      showLine("Key Down: " + mods(flags) + key_name(e));
    else if (e.id == Event.KEY_ACTION)          // функциональная клавиша
      showLine("Function Key Down: " + mods(flags) +
            function_key_name(key));
    return true;
  }
  public boolean keyUp(Event e, int key) {
    int flags = e.modifiers;
    if (e.id == Event.KEY_RELEASE)              // обычная клавиша
      showLine("Key Up: " + mods(flags) + key_name(e));
    else if (e.id == Event.KEY_ACTION_RELEASE)  // функциональная клавиша
```

```
    showLine("Function Key Up: " + mods(flags) +
         function_key_name(key));
  return true;
}

// Оставшиеся методы помогают нам рассортировать
// различные модификаторы и клавиши

// Возвращаем текущий список клавиш-модификаторов
private String mods(int flags) {
  String s = "[ ";
  if (flags == 0) return "";
  if ((flags & Event.SHIFT_MASK) != 0) s += "Shift ";
  if ((flags & Event.CTRL_MASK) != 0) s += "Control ";
  if ((flags & Event.META_MASK) != 0) s += "Meta ";
  if ((flags & Event.ALT_MASK) != 0) s += "Alt ";
  s += "] ";
  return s;
}

// Возвращаем имя обычной (нефункциональной) клавиши.
private String key_name(Event e) {
  char c = (char) e.key;
  // Если установлен флаг CTRL, обрабатываем
  // управляющие символы
  if (e.controlDown()) {
    if (c < ' ') {
      c += '@';
      return "^" + c;
    }
  }
  // Если флаг CTRL не установлен, то определенные
  // управляющие символы ASCII имеют специальное значение
  else {
    switch (c) {
    case '\n': return "Return";
    case '\t': return "Tab";
    case '^[': return "Escape";
    case '^H': return "Backspace";
    }
  }
  // Обрабатываем остальные варианты
  if (c == '^?') return "Delete";
  else if (c == ' ') return "Space";
  else return String.valueOf(c);
}

// Возвращаем имя функциональной клавиши. Просто сравниваем клавишу
// с константами, определенными в классе Event.
private String function_key_name(int key) {
  switch(key) {
  case Event.HOME: return "Home";   case Event.END: return "End";
  case Event.PGUP: return "Page Up"; case Event.PGDN: return"Page Down";
```

```
      case Event.UP: return "Up"; case Event.DOWN: return "Down";
      case Event.LEFT: return "Left";   case Event.RIGHT: return "Right";
      case Event.F1: return "F1"; case Event.F2: return "F2";
      case Event.F3: return "F3"; case Event.F4: return "F4";
      case Event.F5: return "F5"; case Event.F6: return "F6";
      case Event.F7: return "F7"; case Event.F8: return "F8";
      case Event.F9: return "F9"; case Event.F10: return "F10";
      case Event.F11: return "F11"; case Event.F12: return "F12";
      }
      return "Unknown Function Key";
    }

    /** Список строк для отображения в окне */
    protected Vector lines = new Vector();
    /** Добавляем к списку новую строку и отображаем его */
    protected void showLine(String s) {
      if (lines.size() == 20) lines.removeElementAt(0);
      lines.addElement(s);
      repaint();
    }
    /** Этот метод перерисовывает текст в окне */
    public void paint(Graphics g) {
      for(int i = 0; i < lines.size(); i++)
        g.drawString((String)lines.elementAt(i), 20, i*16 + 50);
    }
  }
```

Чтение параметров апплета

В примере 15.5 показан расширенный вариант нашего апплета `Scribble`. Класс `ColorScribble` является производным от `Scribble` и добавляет возможность рисования указанным цветом на фоне указанного цвета.

В классе `ColorScribble` есть метод `init()`, читающий значения двух параметров апплета, которые могут быть указаны с помощью тега `<PARAM>` HTML-файла апплета. Значения типа `String`, возвращаемые методом `getParameter()`, преобразуются в значения цвета и указываются в качестве цвета фона и цвета переднего плана апплета по умолчанию. Заметим, что метод `init()` вызывает метод `init()` базового класса, поэтому последний может проинициализировать себя.

Кроме этого в примере показано использование методов `getAppletInfo()` и `getParameterInfo()`. Эти методы предоставляют текстовую информацию об апплете (его автор, версия, авторское право и т. д.) и параметрах, которые он может принимать (имена параметров, их типы и описания). Обычно апплету следует реализовать эти методы, хотя нынешнее поколение броузеров в действительности даже не использует их. (Однако приложение *appletviewer* из состава JDK вызывает эти методы.)

Пример 15.5. ColorScribble.java

```
package com.davidflanagan.examples.applet;
import java.applet.*;
```

```
import java.awt.*;

/**
 * Версия апплета Scribble; считывает два его параметра для установки цветов
 * фона и переднего плана, а также возвращает по запросу информацию о себе.
 **/
public class ColorScribble extends Scribble {
  // Считываем два параметра и устанавливаем цвета.
  public void init() {
    // Даем возможность базовому классу проинициализировать себя
    super.init();
    Color foreground = getColorParameter("foreground");
    Color background = getColorParameter("background");
    if (foreground != null) this.setForeground(foreground);
    if (background != null) this.setBackground(background);
  }

  // Считываем указанный параметр. Интерпретируем его как шестнадцатеричное
  // число в форме RRGGBB и преобразуем его в цвет.
  protected Color getColorParameter(String name) {
    String value = this.getParameter(name);
    try { return new Color(Integer.parseInt(value, 16)); }
    catch (Exception e) { return null; }
  }

  // Возвращаем информацию, пригодную для отображения
  // в диалоговом окне About.
  public String getAppletInfo() {
    return "ColorScribble v. 0.03. Written by David Flanagan.";
  }

  // Возвращаем информацию о поддерживаемых параметрах.
  // Броузеры и визуализаторы апплетов могут показать эту информацию, а также
  // могут разрешить пользователю установить значения этих параметров.
  public String[][] getParameterInfo() { return info; }

  // Здесь находится информация, возвращаемая методом getParameterInfo().
  // Это массив массивов строк, описывающих каждый параметр.
  // Формат: имя параметра, тип параметра и описание параметра
  private String[][] info = {
    {"foreground", "hexadecimal color value", "foreground color"},
    {"background", "hexadecimal color value", "background color"}
  };
}
```

Следующий фрагмент HTML-файла ссылается на наш апплет и демонстрирует способ задания значений параметров с помощью тега `<PARAM>`**.**

```
<applet code="com.davidflanagan.examples.applet.ColorScribble.class"
codebase="../../../../"width=400 height=400>
 <param name="foreground" value="FF0000">
 <param name="background" value="CCFFCC">
</applet>
```

Изображения и звук

В примере 15.6 показан Java-апплет, реализующий на стороне клиента простую карту изображений (image map), способную подсвечивать активный участок изображения («hot spot») и воспроизводить звуковой клип, когда пользователь щелкает на изображении.

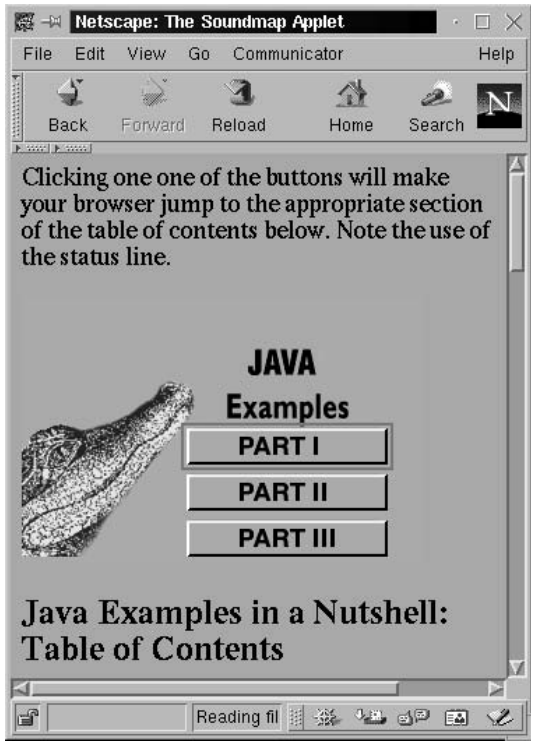


Рис. 15.3. *Апплет imagemap*

Этот апплет демонстрирует несколько важных приемов, используемых при создании апплетов:

- Метод `getParameter()` ищет имя изображения для показа и аудиоклипа для воспроизведения, когда пользователь щелкает мышью. Кроме того, он считывает список прямоугольников и URL, задающих активные участки (hot spots) и гиперссылки карты изображений.
- Методы `getImage()` и `getDocumentBase()` загружают в методе `init()` изображение (объект `Image`), а `Graphics.drawImage()` выводит изображения в методе `paint()`.
- Метод `getAudioClip()` загружает в методе `init()` звуковой файл (объект `AudioClip`), а `AudioClip.play()` проигрывает его в методе `mousePressed()`.

- **События обрабатываются с помощью модели событий Java 1.1. Метод** `showStatus()` **выводит URL назначения, когда пользователь нажимает кнопку мыши на активном участке карты, а метод** `AppletContext.showDocument()` **заставляет броузер показать этот URL, когда пользователь отпускает кнопку мыши.**
- **Отдельные активные участки представлены экземплярами внутреннего класса** `ImagemapRectangle`**. Класс** `java.util.Vector` **хранит список объектов «hot-spot», а** `java.util.StringTokenizer` **анализирует описания этих участков.**

В следующем фрагменте HTML-файла *Soundmap.html* показан пример свойств, считываемых этим апплетом.

```
<APPLET code="com/davidflanagan/examples/applet/Soundmap.class"
codebase="../../../../"
width=288 height=288>
 <PARAM name="image" value="java_parts.gif">
 <PARAM name="sound" value="chirp.au">
 <PARAM name="rect0" value="114,95,151,33,#p1">
 <PARAM name="rect1" value="114,128,151,33,#p2">
 <PARAM name="rect2" value="114,161,151,33,#p3">
 <PARAM name="rect3" value="114,194,151,33,#p4">
 <PARAM name="rect4" value="114,227,151,33,#p5">
</APPLET>
```

Пример 15.6. Soundmap.java

```
package com.davidflanagan.examples.applet;
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.util.*;

/**
 * Java-апплет, имитирующий карту изображений на стороне клиента.
 * Воспроизводит звук каждый раз, когда пользователь нажимает
 * одну из гиперссылок.
 */
public class Soundmap extends Applet implements MouseListener {
 protected Image image;      // Показываемое изображение.
 protected Vector rects;     // Список прямоугольников.
 protected AudioClip sound; // Воспроизводимый звук
 protected ImagemapRectangle highlight; // Подсвеченный прямоугольник

 /** Инициализация апплета */
 public void init() {
  // Ищем имя изображения относительно базового URL и загружаем его.
  // Обратите внимание на использование в одной строке трех методов апплета.
  image = this.getImage(this.getDocumentBase(),
        this.getParameter("image"));

  // Ищем и анализируем список прямоугольных участков и их URL.
```

```
 // Вспомогательная программа getRectangleParameter() определена ниже.
 rects = new Vector();
 ImagemapRectangle r;
 for(int i = 0; (r = getRectangleParameter("rect" + i)) != null; i++)
  rects.addElement(r);

 // Ищем звук, воспроизводимый, когда пользователь
 // выбирает один из этих участков.
 sound = this.getAudioClip(this.getDocumentBase(),
        this.getParameter("sound"));

 // Указываем объект-слушатель, отвечающий на события
 // нажатия и отпускания кнопок мыши. Обратите внимание,
 // что здесь используется модель событий Java 1.1.
 this.addMouseListener(this);
}

/**
 * Вызывается при выгрузке апплета из системы.
 * Здесь мы освобождаем уже ненужное изображение.
 * Это позволяет быстрее освободить память и другие ресурсы.
 **/
public void destroy() { image.flush(); }

/**
 * Для отображения апплета мы просто рисуем изображение
 * и выделяем текущий прямоугольник, если такой есть.
 **/
public void paint(Graphics g) {
 g.drawImage(image, 0, 0, this);
 if (highlight != null) {
  g.setColor(Color.red);
  g.drawRect(highlight.x, highlight.y,
     highlight.width, highlight.height);
  g.drawRect(highlight.x+1, highlight.y+1,
     highlight.width-2, highlight.height-2);
 }
}

/**
 * Мы замещаем этот метод для того, чтобы он не очищал фон перед вызовом
 * метода paint(). Эта очистка не обязательна, т. к. paint() перезаписывает
 * с помощью изображения все пространство. Данный прием уменьшает мерцание.
 **/
public void update(Graphics g) { paint(g); }

/**
 * Анализируем разделенный запятыми список координат
 * прямоугольников и URL. Результат используется для чтения
 * описаний прямоугольников карты изображений из параметров апплета.
 **/
protected ImagemapRectangle getRectangleParameter(String name) {
 int x, y, w, h;
 URL url;
```

```
 String value = this.getParameter(name);
 if (value == null) return null;

 try {
  StringTokenizer st = new StringTokenizer(value, ",");
  x = Integer.parseInt(st.nextToken());
  y = Integer.parseInt(st.nextToken());
  w = Integer.parseInt(st.nextToken());
  h = Integer.parseInt(st.nextToken());
  url = new URL(this.getDocumentBase(), st.nextToken());
 }
 catch (NoSuchElementException e) { return null; }
 catch (NumberFormatException e) { return null; }
 catch (MalformedURLException e) { return null; }

 return new ImagemapRectangle(x, y, w, h, url);
}

/** Этот метод вызывается при нажатии кнопки мыши. */
public void mousePressed(MouseEvent e) {
 // При нажатии кнопки проверяем, находимся ли мы
 // внутри прямоугольника. Если это так, подсвечиваем
 // прямоугольник, выводим сообщение и воспроизводим звук.
 // Вспомогательный метод findrect() определен ниже.
 ImagemapRectangle r = findrect(e);
 // Если прямоугольник найден и еще не подсвечен
 if (r != null && r != highlight) {
  highlight = r;               // Запоминаем этот прямоугольник
  showStatus("To: " + r.url);  // отображаем его URL в строке состояния
  sound.play();                // воспроизводим звук
  repaint();                   // для подсвечивания запрашиваем перерисовку
 }
}

/** Этот метод вызывается при отпускании кнопки мыши. */
public void mouseReleased(MouseEvent e) {
 // Если пользователь отпускает кнопку мыши над выделенным
 // прямоугольником, командуем броузеру показать его URL.
 // Кроме того, снимаем выделение (подсветку) и очищаем строку состояния
 if (highlight != null) {
  ImagemapRectangle r = findrect(e);
  if (r == highlight) getAppletContext().showDocument(r.url);
  showStatus("");    // очищаем сообщение
  highlight = null;  // забываем о подсветке
  repaint();         // запрашиваем перерисовку
 }
}

/** Неиспользуемые методы интерфейса MouseListener */
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
public void mouseClicked(MouseEvent e) {}

/** Ищем прямоугольник, в котором мы находимся. */
```

```
 protected ImagemapRectangle findrect(MouseEvent e) {
  int i, x = e.getX(), y = e.getY();
  for(i = 0; i < rects.size(); i++) {
   ImagemapRectangle r = (ImagemapRectangle) rects.elementAt(i);
   if (r.contains(x, y)) return r;
  }
  return null;
 }

 /**
  * Вспомогательный класс. Похож на java.awt.Rectangle,
  * но с полем URL. Обратите внимание на использование
  * вложенного класса верхнего уровня.
  **/
 static class ImagemapRectangle extends Rectangle {
  URL url;
  public ImagemapRectangle(int x, int y, int w, int h, URL url) {
   super(x, y, w, h);
   this.url = url;
  }
 }
}
```

Файлы JAR

Созданному в предыдущем разделе апплету `Soundmap` требуются для его работы четыре файла: файл класса для самого апплета, файл класса для содержащихся в нем вложенных классов, файл изображения и звуковой файл. Он может быть загружен с помощью тега `<APPLET>` наподобие следующего:

```
<APPLET code="com/davidflanagan/examples/applet/Soundmap.class"
codebase="../../../../" width=288 height=288>
 ...
</APPLET>
```

Однако при загрузке апплета таким способом каждый из четырех упомянутых файлов передается в несжатом виде с использованием отдельного HTML-запроса. Как вы догадались, это крайне неэффективно.

Вместо этого, как и в Java 1.1, вы можете объединить эти четыре файла в один JAR-файл. Этот единый сжатый файл (это ZIP-файл) может быть передан с веб-сервера броузеру гораздо более эффективно. Для создания файла JAR используйте утилиту *jar*, имеющую схожий синтаксис с командой *tar* системы Unix.

```
% jar cf applets.jar com/davidflanagan/examples/applet
```

Эта команда создает новый файл *applets.jar*, содержащий все файлы из каталога *com/davidflanagan/examples/applet*.

Для того чтобы использовать файл JAR, вы должны указать его в качестве значения атрибута ARCHIVE тега <APPLET>.

```
<APPLET code="com/davidflanagan/examples/applet/Soundmap.class"
archive="applets.jar" width=288 height=288>
 ...
</APPLET>
```

Обратите внимание, что атрибут ARCHIVE не заменяет атрибут CODE. ARCHIVE указывает, где нужно искать файлы, но CODE все еще требуется для указания броузеру, какой файл архива является файлом класса апплета, подлежащего запуску. На практике в атрибуте ARCHIVE может быть указан список разделенных запятой JAR-файлов. Броузер или визуализатор апплетов будет просматривать эти архивы в поиске файлов, необходимых апплету. Однако если какой-либо файл не найден в архиве, броузер использует старый механизм и пытается загрузить этот файл с веб-сервера с использованием отдельного HTTP-запроса.

Упражнения

15-1. Доработайте апплет FirstApplet так, чтобы использовать возможности, предлагаемые классом Graphics, для вывода сообщения «Hello World» более визуально интересным способом.

15-2. Измените апплет Clock так, чтобы его можно было конфигурировать через его параметры. Ваш модифицированный апплет должен считывать параметры, указывающие частоту обновления таймера, а также шрифт, цвет и формат представления времени. (См. класс java.text.SimpleDateFormat для изучения механизма форматирования даты и времени в соответствии с заданным шаблоном.) Кроме того, добавьте метод getParameterInfo(), описывающий параметры апплета, которые вы определили, и измените метод getAppletInfo(), включив сообщение, описывающее ваше участие в создании апплета.

15-3. Модифицируйте апплет Soundmap так, чтобы он реагировал на события перемещения мыши так же, как на события нажатия и отпускания кнопок мыши. Ваш модифицированный апплет должен выделять каждый прямоугольник, над которым перемещается мышь, даже если у нее не нажата ни одна кнопка.

15-4. Напишите апплет, отображающий произвольный набор линий, описанных в одном или нескольких параметрах апплета. Протестируйте ваш апплет, заставив его рисовать что-нибудь интересное. Вы можете заглянуть в пример 13.2, чтобы вспомнить способ кодировки набора отрезков линий в текстовой форме, пригодной для использования в качестве параметров апплета.

III

Enterprise Java

Часть III содержит примеры, иллюстрирующие ключевые Java API, используемые в корпоративных проектах. Эти примеры соответствуют той части языка, которая была рассмотрена в книге «Java Enterprise in a Nutshell».

Глава 16

Вызов удаленных методов (RMI)

В этой главе представлены примеры использования технологии вызова удаленных методов (Remote Method Invocation, RMI) реализованной в пакетах `java.rmi` и `java.rmi.server`. Вызов удаленных методов является мощной технологией для разработки сетевых приложений, освобождающей программиста от необходимости заботиться о деталях реализации сетевых соединений на нижнем уровне. RMI позволяет перейти от модели вычислений типа клиент–сервер к более общей модели удаленных объектов. В этой модели сервер определяет объекты, которые могут использоваться удаленными клиентами. Клиенты вызывают методы удаленных объектов так же, как если бы они были локальными объектами, выполняющимися внутри той же виртуальной машины, что и клиент. Технология RMI скрывает лежащий в ее основе механизм транспортировки параметров методов и возвращаемых значений через сеть. Параметр и возвращаемое значение могут быть либо значением примитивного типа, либо любым сериализуемым объектом.

Для того чтобы создать приложение на базе RMI, вы должны следовать приведенным ниже инструкциям:

- Создайте интерфейс, расширяющий `java.rmi.Remote`. В этом интерфейсе определены экспортируемые методы, реализуемые удаленными объектами (то есть методы, реализуемые сервером и вызываемые удаленным клиентом). Каждый метод этого интерфейса должен быть объявлен как генерирующий исключение `java.rmi.RemoteException`, которое является базовым классом многих более специфичных классов исключений RMI. Каждый удаленный метод должен объявлять, что он может сгенерировать `RemoteException` из-за того, что существуют некоторые ситуации, приводящие к возникновению ошибок во время процесса вызова удаленных методов через сеть.

- Определите класс, производный от `java.rmi.server.UnicastRemoteObject` (или от какого-нибудь связанного с ним), реализующий ваш удаленный интерфейс. Этот класс представляет удаленный, или серверный, объект. Кроме объявления того, что его удаленные методы генерируют исключения `RemoteException`, удаленный объект не должен делать что-либо особенное, чтобы позволить вызывать его методы удаленно. Объект `UnicastRemoteObject` и вся остальная часть RMI-инфраструктуры обрабатывают это автоматически.

- Напишите программу (сервер), создающую экземпляр вашего удаленного объекта. Экспортируйте объект, сделав его доступным для использования клиентами путем регистрации имени объекта в службе реестра. Как правило, это выполняется с помощью класса `java.rmi.Naming` и программы *rmiregistry*. Кроме того, серверная программа может сама выполнять функции сервера реестра, используя класс `LocateRegistry` и интерфейс `Registry` из пакета `java.rmi.registry`.

- После того как вы с помощью *javac* откомпилируете серверную программу, воспользуйтесь утилитой *rmic*, чтобы сгенерировать для удаленного объекта заглушку (stub) и каркас (skeleton). При использовании RMI клиент и сервер не взаимодействуют непосредственно. На стороне клиента ссылка на удаленный объект реализуется в виде экземпляра класса заглушки. Когда клиент вызывает удаленный метод, в действительности вызывается метод объекта-заглушки. Заглушка производит необходимые сетевые операции по передаче этого вызова находящемуся на сервере классу каркаса. Этот каркас транслирует пришедший по сети запрос в вызов метода серверного объекта, а затем передает возвращенное им значение обратно заглушке, которая, в свою очередь, возвращает его клиенту. Все это представляет собой довольно сложную систему, но, к счастью, прикладным программистам никогда не приходится думать о заглушках и каркасах; они генерируются автоматически утилитой *rmic*. Вызовите из командной строки *rmic*, указав имя удаленного класса объекта (не интерфейса). Она создаст и откомпилирует два новых класса с суффиксами `_Stub` и `_Skel`.

- Если сервер использует службу реестра по умолчанию, предоставленную классом `Naming`, вы должны запустить сервер реестра, если он еще не запущен. Вы можете запустить этот сервер путем вызова программы *rmiregistry*.

- Сейчас вы можете написать клиентскую программу, использующую экспортированный сервером удаленный объект. Прежде всего, клиенту необходимо получить ссылку на удаленный объект, используя класс `Naming` для поиска объекта по имени. Это имя обычно задается в форме *rmi*:URL. Удаленная ссылка, которая будет возвращена, представляет собой экземпляр интерфейса `Remote` объекта (или, более точно, объект-заглушку удаленного объекта). Как толь-

ко клиент получил этот удаленный объект, он может вызывать его методы точно так же, как он вызывал бы методы локального объекта. Он должен только знать, что все удаленные методы могут генерировать объекты исключений `RemoteException` и что при возникновении сетевых ошибок это может случаться в самый неожиданный момент.

- Наконец, запустите и серверную программу, и клиент!

В следующих разделах этой главы приведены два законченных примера использования RMI, которые проходят отмеченные здесь этапы. Первый пример является сравнительно простой программой удаленного банковского обслуживания, тогда как второй пример – это сложная и объемная система многопользовательской области (multiuser domain, MUD).

Удаленное банковское обслуживание

В примере 16.1 показан класс `Bank`, который содержит внутренние классы и интерфейсы для примера клиент-серверной банковской системы. В этом примере в интерфейсе `RemoteBank` определены удаленные методы для открытия и закрытия счета, внесения и снятия денег, проверки баланса счета и получения выписки проводок, связанных со счетом. Класс `Bank` содержит все необходимые для данного примера классы и интерфейсы, за исключением серверного класса – класса, который в действительности реализует интерфейс `RemoteBank`. Этот серверный класс приведен в примере 16.2.

В примере 16.1 определены следующие внутренние классы и интерфейсы:

`RemoteBank`

Удаленный интерфейс, реализуемый сервером банка и используемый клиентом банка.

`FunnyMoney`

Простейший класс, представляющий деньги этого примера банковского обслуживания. Он является простой оберткой вокруг типа `int`, но служит для демонстрации того, как сериализуемые объекты могут быть переданы в качестве параметров удаленных методов и возвращены ими.

`BankingException`

Простой подкласс исключения, который представляет исключения, имеющие отношение к банковскому делу, такие как «недостаток денежных средств». Он демонстрирует, как реализации удаленных методов, находящиеся на сервере, могут генерировать исключения, которые затем будут переданы по сети и возбуждены в клиентской программе.

`Client`

Этот класс является автономной программой, выполняющей функции простого клиента банковского сервера. Он использует метод `Naming.lookup()` для поиска в системном реестре нужного ему объекта `RemoteBank` и последующего вызова различных методов этого объекта в зависимости от параметров его командной строки. Все это на самом деле очень просто, использование RMI почти прозрачно.

Сеанс использования класса `Bank.Client` может выглядеть так (обратите внимание, что параметр командной строки «david» – это имя счета, а «javanut» – защитный пароль счета):

```
% java com.davidflanagan.examples.rmi.Bank\$Client open david javanut
Счет открыт.
% java com.davidflanagan.examples.rmi.Bank\$Client deposit david javanut 1000
Внесено 1000 деревянных никелей.
% java com.davidflanagan.examples.rmi.Bank\$Client withdraw david javanut 100
Снято 100 деревянных никелей.
% java com.davidflanagan.examples.rmi.Bank\$Client balance david javanut
У вас в банке 900 деревянных никелей.
% java com.davidflanagan.examples.rmi.Bank\$Client history david javanut
Счет открыт в Срд 12 Июл 2000 15:30:12
Внесено 1000 в Срд 12 Июл 2000 15:30:31
Снято 100 в Срд 12 Июл 2000 15:30:39
% java com.davidflanagan.examples.rmi.Bank\$Client close david javanut
Вам возвращено 900 деревянных никелей.
Большое спасибо.
```

В этом примере сеанса клиент банка выполняется на том же хосте, что и сервер. Хотя в этом и нет необходимости, класс `Client` ищет системное свойство с именем `bank`, чтобы определить, с каким банковским сервером ему нужно соединяться. Таким образом, вы можете вызывать клиентскую программу следующим образом (одна длинная командная строка была разбита на две строки):

```
% java -Dbank=rmi://bank.trustme.com/TrustyBank \
com.davidflanagan.examples.rmi.Bank\$Client open david javanut
```

Пример 16.1. Bank.java

```
package com.davidflanagan.examples.rmi;
import java.rmi.*;
import java.util.List;

/**
 * Данный класс является контейнером, просто содержащим
 * другие классы и интерфейсы для удаленной банковской системы
 **/
public class Bank {
  /**
   * Это интерфейс, в котором определены методы,
   * экспортируемые банковским сервером.
   **/
```

```
public interface RemoteBank extends Remote {
  /** Открываем новый счет с указанным  именем и паролем */
  public void openAccount(String name, String password)
    throws RemoteException, BankingException;

  /** Закрываем именной счет */
  public FunnyMoney closeAccount(String name, String password)
    throws RemoteException, BankingException;

  /** Вносим деньги на именной счет */
  public void deposit(String name, String password, FunnyMoney money)
    throws RemoteException, BankingException;

  /** Снимаем заданную денежную сумму с именного счета */
  public FunnyMoney withdraw(String name, String password, int amount)
    throws RemoteException, BankingException;

  /** Возвращаем сумму, хранящуюся на именном счете */
  public int getBalance(String name, String password)
    throws RemoteException, BankingException;

  /**
   * Возвращаем список строк, содержащий выписку проводок именного счета
   **/
  public List getTransactionHistory(String name, String password)
    throws RemoteException, BankingException;
}

/**
 * Этот простой класс представляет денежный счет. Данная реализация
 * представляет собой простую обертку вокруг целого числа.
 * Он полезен для демонстрации того, как RMI может принимать в качестве
 * аргументов произвольные нестроковые объекты и возвращать их
 * в качестве значений, если они являются сериализуемыми.
 * Более сложная реализация этого класса FunnyMoney может
 * включать серийный номер, цифровую подпись и другие средства
 * безопасности для обеспечения его уникальности и защиты.
 **/
public static class FunnyMoney implements java.io.Serializable {
  public int amount;
  public FunnyMoney(int amount) { this.amount = amount; }
}

/**
 * Это тип исключения, используемого для представления
 * исключительных ситуаций, возникающих в банковском деле,
 * таких как "Недостаток денежных средств" и "Неверный пароль"
 **/
public static class BankingException extends Exception {
  public BankingException(String msg) { super(msg); }
}

/**
 * Этот класс является простой автономной клиентской
```

```
 * программой, взаимодействующей с сервером RemoteBank.
 * Она вызывает различные методы RemoteBank в зависимости
 * от параметров ее командной строки и демонстрирует, как просто
 * при помощи RMI реализуется взаимодействие с сервером.
 **/
public static class Client {
  public static void main(String[] args) {
    try {
      // Выясняем, с каким RemoteBank нужно соединяться,
      // путем чтения системного свойства (указанного
      // в командной строке с ключом -D) или, если оно
      // не задано, используем URL по умолчанию. Заметьте,
      // что по умолчанию клиент пытается соединиться
      // с сервером, расположенным на локальной машине
      String url = System.getProperty("bank", "rmi://FirstRemote");

      // Ищем этот сервер RemoteBank, используя объект
      // Naming, связанный с сервером rmiregistry.
      // По заданному url этот вызов возвращает объект
      // RemoteBank, методы которого могут быть вызваны удаленно
      RemoteBank bank = (RemoteBank) Naming.lookup(url);

      // Преобразуем команду пользователя к нижнему регистру
      String cmd = args[0].toLowerCase();

      // Проверяем команду на совпадение с возможными значениями
      if (cmd.equals("open")) {      // Открываем счет
        bank.openAccount(args[1], args[2]);
        System.out.println("Счет открыт.");
      }
      else if (cmd.equals("close")) {   // Закрываем счет
        FunnyMoney money = bank.closeAccount(args[1], args[2]);
        // Наша валюта называется “деревянный никель”
        System.out.println("Вам возвращено "+money.amount +
                  " деревянных никелей.");
        System.out.println("Большое спасибо.");
      }
      else if (cmd.equals("deposit")) {  // Внесение денег
        FunnyMoney money=new FunnyMoney(Integer.parseInt(args[3]));
        bank.deposit(args[1], args[2], money);
        System.out.println("Внесено " + money.amount +
                           " деревянных никелей.");
      }
      else if (cmd.equals("withdraw")) { // Снятие денег
        FunnyMoney money = bank.withdraw(args[1], args[2],
                        Integer.parseInt(args[3]));
       System.out.println("Снято " + money.amount + " деревянных никелей.");
      }
      else if (cmd.equals("balance")) {  // Проверка баланса счета
        int amt = bank.getBalance(args[1], args[2]);
       System.out.println("У вас в банке " + amt + " деревянных никелей.");
      }
```

```
            else if (cmd.equals("history")) {  // Получение выписки проводок
              List transactions = bank.getTransactionHistory(args[1], args[2]);
              for(int i = 0; i < transactions.size(); i++)
                System.out.println(transactions.get(i));
            }
            else System.out.println("Неизвестная команда");
          }
          // Перехватываем и выводим исключения RMI
          catch (RemoteException e) { System.err.println(e); }
          // Перехватываем и выводим исключения, относящиеся к банковскому делу
          catch (BankingException e) { System.err.println(e.getMessage()); }
          // Другие исключения - это, вероятно, синтаксические ошибки
          // пользователя, поэтому показываем строку формата.
          catch (Exception e) {
            System.err.println(e);
            System.err.println("Формат: java [-Dbank=<url>] Bank$Client " +
                       "<cmd> <name> <password> [<amount>]");
            System.err.println("где cmd это: open, close, deposit, "
               + "withdraw, balance, history");
          }
        }
      }
    }
```

Банковский сервер

В примере 16.1 были определены интерфейс `RemoteBank` и банковская клиентская программа. В примере 16.2 приведен класс `RemoteBankServer`, реализующий интерфейс `RemoteBank` и действующий в качестве сервера для программы `Bank.Client`. В этот класс включен метод `main()`, поэтому он может выполняться как автономная программа. Этот метод создает объект `RemoteBankServer` и регистрирует его с помощью метода `Naming.rebind()`, чтобы клиенты смогли найти его. Он считывает системное свойство `bankname`, чтобы определить имя, которое нужно использовать для регистрации банка, но по умолчанию использует имя `FirstRemote`. (Это то же имя, которое `Bank.Client` использует по умолчанию.)

`RemoteBankServer` реализует интерфейс `RemoteBank`, то есть он предоставляет реализацию всех удаленных методов, определенных в данном интерфейсе. Также в нем определено несколько служебных методов, которые сами не являются удаленными, но используются ими. Заметим, что в состав `RemoteBankServer` включен внутренний класс `Account`, который сохраняет всю информацию, относящуюся к отдельному банковскому счету. Он содержит хеш-таблицу для сопоставления имен счетов с объектами `Account`. Различные удаленные методы выполняют поиск именных счетов, проверяют пароль и производят определенные манипуляции со счетом. Любой удаленный объект RMI должен уметь обрабатывать множественные параллельные вызовы методов, потому что в одно и то же время данный объект может использоваться несколькими

клиентами. Чтобы предотвратить открытие, закрытие или изменение одного счета одновременно двумя клиентами, RemoteBankServer использует синхронизированные методы и операторы synchronized.

Прежде чем вы сможете выполнить программу RemoteBankServer, нужно откомпилировать ее, сгенерировать классы заглушки и каркаса и запустить сервер *rmiregistry* (если он еще не запущен). Вы можете сделать все это при помощи следующих команд (в системе Unix). Обратите внимание на параметр -d утилиты *rmic*: он сообщает компилятору RMI, куда поместить классы заглушки и каркаса. Допустим, что файл RemoteBankServer.class находится в текущем каталоге, тогда при приведенном здесь способе использования сгенерированные классы будут помещены в тот же каталог.

```
% javac RemoteBankServer.java
% rmic -d ../../../../ com.davidflanagan.examples.rmi.RemoteBankServer
% rmiregistry &
% java com.davidflanagan.examples.rmi.RemoteBankServer
FirstRemote открыт и ждет клиентов.
```

Обратите внимание, что пример 16.2 содержит фатальный недостаток: при сбое сервера банка все данные банковских счетов будут утрачены, что, вероятно, приведет клиентов в ярость! В главе 17 приведена другая реализация интерфейса RemoteBank. Она использует базу данных для хранения данных о счетах более надежным способом.

Пример 16.2. RemoteBankServer.java

```
package com.davidflanagan.examples.rmi;
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
import Bank.*;

/**
 * Этот класс реализует удаленные методы, определенные
 * в интерфейсе RemoteBank. Хотя у него есть серьезный недостаток:
 * при выключении сервера все данные о счете будут потеряны
 **/
public class RemoteBankServer extends UnicastRemoteObject implements RemoteBank
{
 /**
  * Этот вложенный класс хранит данные об отдельном банковском счете.
  **/
 class Account {
  String password;                    // пароль счета
  int balance;                        // баланс счета
  List transactions = new ArrayList(); // история проводок счета
  Account(String password) {
   this.password = password;
   transactions.add("Счет открыт " + new Date());
  }
```

```
}

/**
 * Эта хеш-таблица хранит все открытые счета и связывает
 * имя счета с объектом Account
 **/
Map accounts = new HashMap();

/**
 * Этот конструктор ничего не делает, но поскольку конструктор
 * базового класса генерирует исключение, это исключение
 * должно быть объявлено и здесь.
 **/
public RemoteBankServer() throws RemoteException { super(); }

/**
 * Открываем банковский счет с указанным именем и паролем
 * Этот метод сделан синхронизированным для обеспечения
 * потоковой безопасности, так как в нем происходит
 * манипулирование хеш-таблицей счетов.
 **/
public synchronized void openAccount(String name, String password)
 throws RemoteException, BankingException
{
 // Проверяем, есть ли уже счет под таким именем
 if (accounts.get(name) != null)
  throw new BankingException("Счет уже существует.");
 // Иначе он не существует, поэтому создаем
 Account acct = new Account(password);
 // и регистрируем его.
 accounts.put(name, acct);
}

/**
 * Этот внутренний метод не является удаленным. Принимая имя и пароль,
 * он проверяет, не существует ли уже счет с таким именем и паролем.
 * Если это так, он возвращает объект Account.
 * Иначе - генерирует исключение.
 **/
Account verify(String name, String password) throws BankingException {
 synchronized(accounts) {
  Account acct = (Account)accounts.get(name);
  if (acct == null) throw new BankingException("Нет такого счета");
  if (!password.equals(acct.password))
   throw new BankingException("Неверный пароль");
  return acct;
 }
}

/**
 * Закрываем указанный счет. Этот метод сделан синхронизированным
 * для обеспечения потоковой безопасности, так как в нем
 * происходит манипулирование хеш-таблицей счетов
 **/
```

```
public synchronized FunnyMoney closeAccount(String name, String password)
 throws RemoteException, BankingException
{
 Account acct;
 acct = verify(name, password);
 accounts.remove(name);
 // Перед изменением баланса или проведением проводок по любому счету
 // ради потоковой безопасности мы, прежде всего, должны получить
 // блокировку этого счета.
 synchronized (acct) {
  int balance = acct.balance;
  acct.balance = 0;
  return new FunnyMoney(balance);
 }
}

/** Вносим заданную сумму FunnyMoney на указанный счет */
public void deposit(String name, String password, FunnyMoney money)
 throws RemoteException, BankingException
{
 Account acct = verify(name, password);
 synchronized(acct) {
  acct.balance += money.amount;
  acct.transactions.add("Внесено " + money.amount + " в " + new Date());
 }
}

/** Снимаем заданную сумму с указанного счета */
public FunnyMoney withdraw(String name, String password, int amount)
 throws RemoteException, BankingException
{
 Account acct = verify(name, password);
 synchronized(acct) {
  if (acct.balance < amount)
   throw new BankingException("Недостаточно средств");
  acct.balance -= amount;
  acct.transactions.add("Снято " + amount + " в " + new Date());
  return new FunnyMoney(amount);
 }
}

/** Возвращаем текущий баланс для указанного счета */
public int getBalance(String name, String password)
 throws RemoteException, BankingException
{
 Account acct = verify(name, password);
 synchronized(acct) { return acct.balance; }
}

/**
 * Возвращаем вектор строк, содержащий историю проводок по указанному счету
 **/
public List getTransactionHistory(String name, String password)
```

```
    throws RemoteException, BankingException
  {
    Account acct = verify(name, password);
    synchronized(acct) { return acct.transactions; }
  }

  /**
   * Главная программа, выполняющая этот RemoteBankServer.
   * Создаем объект RemoteBankServer и присваиваем ему имя в реестре.
   * Считываем системное свойство для определения имени, а в качестве имени
   * по умолчанию используем имя FirstRemote. Это все, что необходимо
   * для настройки сервиса. Обо всем остальном позаботится RMI.
   **/
  public static void main(String[] args) {
    try {
      // Создаем объект банковского сервера
      RemoteBankServer bank = new RemoteBankServer();
      // Выясняем, как назвать его
      String name = System.getProperty("bankname", "FirstRemote");
      // Назовем его так
      Naming.rebind(name, bank);
      // Сообщаем миру, что мы готовы к работе
      System.out.println(name + " открыт и ждет клиентов.");
    }
    catch (Exception e) {
      System.err.println(e);
      System.err.println("Формат: java [-Dbankname=<name>] " +
          "com.davidflanagan.examples.rmi.RemoteBankServer");
      // Выходим принудительно из-за возможного существования потоков RMI.
      System.exit(1);
    }
  }
}
```

Многопользовательская область

Многопользовательская область, или MUD (multiuser domain), представляет собой программу (сервер), дающую многим людям (клиентам) возможность взаимодействовать друг с другом и с общим виртуальным окружением. Окружением обычно является ряд комнат, или мест, связанных друг с другом разнообразными выходами (exit). Каждая комната, или место, имеет текстовое описание, служащее в качестве декорации и задающее тон взаимодействия между пользователями. Многие из ранних MUD были составлены из темниц с описаниями комнат, отражающими темную, подземную натуру этого воображаемого мира. Фактически сокращение MUD первоначально означало «многопользовательская темница» (multiuser dungeon). Некоторые MUD служат, прежде всего, в качестве чатов для их клиентов, тогда как другие больше похожи на некую разновидность старых приклю-

ченческих игр, которые сконцентрированы на исследовании окружения и решении головоломок. Другие являются упражнениями в творчестве и групповой динамике, разрешая пользователям добавлять в MUD новые комнаты и элементы.

В примерах с 16.3 по 16.7 приведены классы и интерфейсы, которые определяют простую расширяемую пользователем систему MUD. Такая программа, как этот пример, ясно демонстрирует, насколько парадигма RMI-программирования расширяет модель клиент-сервер. Как мы увидим дальше, `MudServer` и `MudPlace` являются серверными объектами, создающими окружение MUD, внутри которого взаимодействуют пользователи. Но в то же время каждый пользователь, находящийся внутри MUD, представлен удаленным объектом `MudPerson`, который при взаимодействии с другими пользователями действует в качестве сервера. Вместо использования единственного сервера и набора клиентов, эта система действительно является распределенной сетью удаленных объектов, где все взаимодействуют друг с другом. Кто из объектов в действительности является сервером, а кто – клиентом, зависит от вашей точки зрения.

Для того чтобы понять систему MUD, следует дать краткий обзор ее архитектуры. Класс `MudServer` – это простой удаленный объект (и самостоятельная серверная программа), который является точкой входа в MUD и отслеживает имена всех комнат в пределах MUD. Несмотря на свое имя, объект `MudServer` не предоставляет услуг, о которых большинство пользователей думает как о MUD. Это работа класса `MudPlace`.

Каждый объект `MudPlace` представляет отдельную комнату внутри MUD. У каждой комнаты есть имя, описание, список находящихся в ней предметов и людей (пользователей), выходы из этой комнаты и другие комнаты, в которые эти выходы ведут. Выход может вести к смежному объекту `MudPlace`, находящемуся на этом же сервере, или к объекту `MudPlace` другой MUD, расположенной вовсе на другом сервере. Таким образом, MUD-окружение, с которым взаимодействует пользователь, в действительности является сетью объектов `MudPlace`. Описания комнат и предметов и сложность связей между комнатами дают среде MUD то богатство, которое делает ее интересной пользователю.

Каждый пользователь, или человек, в системе MUD представлен объектом `MudPerson`. `MudPerson` – это удаленный объект, имеющий два метода. Один из них возвращает описание человека (то есть то, что видят другие люди, когда они смотрят на этого человека), а второй доставляет человеку (или пользователю, которого представляет `MudPerson`) некоторое сообщение. Эти методы позволяют пользователям смотреть друг на друга и говорить друг с другом. Когда два пользователя встречаются друг с другом в данном `MudPlace` и начинают беседовать, `MudPlace` и сервер, на котором работает MUD, больше не имеют значения. Благодаря мощи RMI два объекта `MudPerson` могут связываться друг с другом напрямую.

Примеры, приведенные ниже, объемны и довольно сложны, но достойны тщательного изучения. Однако, учитывая сложность разрабатываемой MUD-системы, определенные ниже классы и интерфейсы на самом деле удивительно просты. Как вы увидите далее, технология вызовов удаленных методов является очень мощным средством реализации систем такого типа.

Удаленные интерфейсы MUD

В примере 16.3 показан класс `Mud`, играющий роль контейнера внутренних классов и интерфейсов (и одной константы), используемых в оставшейся части MUD-системы. Наиболее важно то, что в `Mud` определено три удаленных интерфейса: `RemoteMudServer`, `RemoteMudPerson` и `RemoteMudPlace`. Они определяют удаленные методы, которые реализуются объектами `MudServer`, `MudPerson` и `MudPlace` соответственно.

Пример 16.3. Mud.java

```
package com.davidflanagan.examples.rmi;
import java.rmi.*;
import java.util.Vector;
import java.io.IOException;

/**
 * В этом классе определено три внутренних удаленных интерфейса, используемых
 * в нашей MUD-игре. Также в нем определена группа подклассов исключений
 * и строковых констант, используемых в качестве префиксов для создания
 * уникальных имен при регистрации MUD-серверов
 **/
public class Mud {
 /**
  * Этот интерфейс определяет экспортируемые методы объекта MUD-сервера
  **/
 public interface RemoteMudServer extends Remote {
  /** Возвращаем имя этой MUD */
  public String getMudName() throws RemoteException;

  /** Возвращаем для этой MUD комнату главного входа */
  public RemoteMudPlace getEntrance() throws RemoteException;

  /** Ищем и возвращаем некоторую другую заданную комнату этой MUD */
  public RemoteMudPlace getNamedPlace(String name)
   throws RemoteException, NoSuchPlace;

  /**
   * Выводим дамп состояния сервера в файл, так чтобы затем
   * он смог восстановить все комнаты, их выходы и предметы, но не людей.
   **/
  public void dump(String password, String filename)
   throws RemoteException, BadPassword, IOException;
 }
```

```
/**
 * Этот интерфейс определяет методы, экспортируемые объектом
 * person, который находится в этой MUD.
 **/
public interface RemoteMudPerson extends Remote {
 /** Возвращаем полное описание человека */
 public String getDescription() throws RemoteException;

 /** Доставляем сообщение для человека */
 public void tell(String message) throws RemoteException;
}

/**
 * Это наиболее важный из всех интерфейсов MUD. В нем определены методы,
 * экспортируемые находящимися внутри MUD местами, или комнатами.
 * У каждой комнаты есть имя и описание, и кроме этого она поддерживает
 * список находящихся в ней людей, предметов и выходов из нее.
 * Здесь имеются методы для получения списков имен этих людей,
 * предметов и выходов. Есть методы для получения объекта RemoteMudPerson
 * для указанного человека, для получения описания указанного предмета
 * и для перехода через заданный выход. Имеются методы для взаимодействия
 * с другими людьми, находящимися в MUD. Есть методы для построения MUD
 * путем создания и уничтожения предметов, добавления новых комнат
 * (и новых выходов из них), для связывания комнат через новые выходы
 * с другими комнатами (возможно, на другом MUD-сервере) и для закрытия
 * существующих выходов.
 **/
public interface RemoteMudPlace extends Remote {
 /** Получаем имя этой комнаты */
 public String getPlaceName() throws RemoteException;

 /** Получаем описание этой комнаты */
 public String getDescription() throws RemoteException;

 /** Получаем имена всех находящихся здесь людей */
 public Vector getNames() throws RemoteException;

 /** Получаем имена всех находящихся здесь вещей */
 public Vector getThings() throws RemoteException;

 /** Получаем имена всех путей отсюда */
 public Vector getExits() throws RemoteException;

 /** Получаем объект RemoteMudPerson для заданного человека. */
 public RemoteMudPerson getPerson(String name)
  throws RemoteException, NoSuchPerson;

 /** Получаем дополнительные сведения об указанной вещи */
 public String examineThing(String name)
  throws RemoteException,NoSuchThing;

 /** Используем указанный выход */
 public RemoteMudPlace go(RemoteMudPerson who, String direction)
  throws RemoteException,NotThere,AlreadyThere,NoSuchExit,LinkFailed;
```

```
/** Посылаем сообщение типа "Давид: Привет всем" */
public void speak(RemoteMudPerson speaker, String msg)
 throws RemoteException, NotThere;

/** Посылаем сообщение типа "Давид громко смеется" */
public void act(RemoteMudPerson speaker, String msg)
 throws RemoteException, NotThere;

/** Добавляем новый предмет в эту комнату */
public void createThing(RemoteMudPerson who, String name,
      String description)
 throws RemoteException, NotThere, AlreadyThere;

/** Удаляем предмет из этой комнаты */
public void destroyThing(RemoteMudPerson who, String thing)
 throws RemoteException, NotThere, NoSuchThing;

/**
 * Создаем новую комнату, связанную с текущей двунаправленным выходом
 **/
public void createPlace(RemoteMudPerson creator,
      String exit, String entrance,
      String name, String description)
 throws RemoteException,NotThere,
   ExitAlreadyExists,PlaceAlreadyExists;

/**
 * Связываем эту комнату (однонаправленно) с некоторой
 * существующей комнатой. Комната назначения может находиться
 * даже на другом сервере.
 **/
public void linkTo(RemoteMudPerson who, String exit,
     String hostname, String mudname, String placename)
 throws RemoteException, NotThere, ExitAlreadyExists, NoSuchPlace;

/** Удаляем существующий выход */
public void close(RemoteMudPerson who, String exit)
 throws RemoteException, NotThere, NoSuchExit;

/**
 * Удаляем этого человека из данной комнаты, оставляя
 * его в неопределенном положении. Посылаем сообщение
 * всем оставшимся в этой комнате.
 **/
public void exit(RemoteMudPerson who, String message)
 throws RemoteException, NotThere;

/**
 * Помещаем человека в новое место, связывая их имена
 * и посылая заданное сообщение всем находящимся в этой комнате.
 * Клиент не должен делать этот метод доступным пользователю.
 * Вместо него пользователи должны применять метод go().
 **/
public void enter(RemoteMudPerson who, String name, String message)
```

```
    throws RemoteException, AlreadyThere;

  /**
   * Возвращаем серверный объект MUD, в котором находится эта комната.
   * Этот метод не должен быть непосредственно видимым для игрока
   **/
  public RemoteMudServer getServer() throws RemoteException;
}

/**
 * Это общий класс исключений, служащий в качестве
 * базового для ряда более специальных типов исключений
 **/
public static class MudException extends Exception {}

/**
 * Эти классы специальных исключений генерируются в различных ситуациях.
 * Имя класса исключения содержит всю необходимую информацию о данном
 * исключении. Эти классы не предоставляют никаких поясняющих сообщений.
 **/
public static class NotThere extends MudException {}
public static class AlreadyThere extends MudException {}
public static class NoSuchThing extends MudException {}
public static class NoSuchPerson extends MudException {}
public static class NoSuchExit extends MudException {}
public static class NoSuchPlace extends MudException {}
public static class ExitAlreadyExists extends MudException {}
public static class PlaceAlreadyExists extends MudException {}
public static class LinkFailed extends MudException {}
public static class BadPassword extends MudException {}

/**
 * Эта константа используется в качестве префикса
 * имени MUD, когда сервер регистрирует данную MUD
 * в реестре RMI, и когда клиент ищет ее в этом реестре.
 * Использование этого префикса помогает предотвратить
 * возможный конфликт имен.
 **/
static final String mudPrefix = "com.davidflanagan.examples.rmi.Mud.";
}
```

Сервер MUD

В примере 16.4 показан класс `MudServer`. Этот класс является автономной программой, которая начинает выполнение MUD. Также она предоставляет реализацию интерфейса `RemoteMudServer`. Как было отмечено выше, объект `MudServer` служит просто входом для MUD: он – не сама MUD. Поэтому он является довольно простым классом. Одна из его наиболее интересных особенностей – это использование классов сериализации `java.io` и классов архивирования `java.util.zip` для сохранения состояния MUD с целью его восстановления позже.

Пример 16.4. MudServer.java

```
package com.davidflanagan.examples.rmi;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.io.*;
import java.util.Hashtable;
import java.util.zip.*;
import com.davidflanagan.examples.rmi.Mud.*;

/**
 * Этот класс реализует интерфейс RemoteMudServer. Также в нем определен
 * метод main(), поэтому вы можете запускать его как самостоятельную
 * программу, которая будет настраивать и инициализировать MUD-сервер.
 * Заметим, что этот MudServer является точкой входа в MUD, но не самой MUD.
 * Наиболее интересная часть функциональных возможностей MUD определяется
 * интерфейсом RemoteMudPlace и реализуется классом RemotePlace.
 * Кроме того, что он является удаленным объектом, этот класс еще
 * и сериализуемый, т. е. состояние MUD может быть записано в файл
 * и в дальнейшем восстановлено. Заметьте, что метод main() предоставляет
 * два способа запуска MUD: первым является запуск "с нуля" с одной
 * начальной комнатой, а второй – восстановление существующей MUD из файла.
 **/
public class MudServer extends UnicastRemoteObject
 implements RemoteMudServer, Serializable
{
 MudPlace entrance; // Стандартный вход в эту  MUD
 String password;   // Пароль, необходимый для создания дампа состояния MUD
 String mudname;    // Имя, под которым зарегистрирована эта MUD
 Hashtable places;  // Соответствие имен комнат и комнат этой MUD

 /**
  * Запускаем  MUD "с нуля", с данными именем и паролем.
  * Создаем в качестве входа начальный объект MudPlace,
  * присваивая ему указанные имя и описание.
  **/
 public MudServer(String mudname, String password,
      String placename, String description)
  throws RemoteException
 {
  this.mudname = mudname;
  this.password = password;
  this.places = new Hashtable();
  // Создаем комнату старта
  try { this.entrance = new MudPlace(this, placename, description); }
  catch (PlaceAlreadyExists e) {} // Не должно происходить
 }

 /** Только для целей сериализации. Никогда ны вызывайте этот конструктор. */
 public MudServer() throws RemoteException {}

 /** Этот удаленный метод возвращает имя MUD */
```

```
public String getMudName() throws RemoteException { return mudname; }

/** Этот удаленный метод возвращает комнату входа в MUD */
public RemoteMudPlace getEntrance() throws RemoteException {
 return entrance;
}

/**
 * Этот удаленный метод возвращает объект RemoteMudPlace
 * для указанной комнаты. В этом смысле MudServer действует
 * как объект реестра RMI, возвращая удаленные объекты, заданные
 * по имени. Проще сделать это путем использования настоящего
 * объекта Registry. Если указанная комната не существует,
 * он генерирует исключение NoSuchPlace.
 **/
public RemoteMudPlace getNamedPlace(String name)
   throws RemoteException, NoSuchPlace
{
 RemoteMudPlace p = (RemoteMudPlace) places.get(name);
 if (p == null) throw new NoSuchPlace();
 return p;
}

/**
 * Задаем имя комнаты, используемое механизмом связывания,
 * реализованным с помощью х'ш-таблицы. Это не удаленный
 * метод. Конструктор MudPlace() вызывает его для
 * регистрации вновь созданной им комнаты.
 **/
public void setPlaceName(RemoteMudPlace place, String name)
 throws PlaceAlreadyExists
{
 if (places.containsKey(name)) throw new PlaceAlreadyExists();
 places.put(name, place);
}

/**
 * Этот удаленный метод выполняет сериализацию и сжатие состояния MUD
 * в заданный файл, если указанный пароль совпадает с тем, который был
 * указан при начальном создании MUD. Заметьте, что в состояние MUD
 * входят все комнаты MUD со всеми предметами и со всеми выходами из них.
 * Люди, находящиеся в MUD, не являются частью его сохраняемого состояния.
 **/
public void dump(String password, String f)
 throws RemoteException, BadPassword, IOException
{
 if ((this.password != null) && !this.password.equals(password))
  throw new BadPassword();
 ObjectOutputStream out = new ObjectOutputStream(
       new GZIPOutputStream(new FileOutputStream(f)));
 out.writeObject(this);
 out.close();
}
```

```
    /**
     * Этот метод main() определяет самостоятельную программу,
     * запускающую MUD-сервер. При вызове с единственным аргументом она считает
     * этот аргумент именем файла, содержащего сериализованное и сжатое
     * состояние существующей MUD, и восстанавливает ее. В противном случае
     * она ожидает получить в командной строке четыре аргумента:
     * имя MUD, пароль, имя комнаты входа в эту MUD и описание этой комнаты.
     * Помимо создания объекта MudServer программа устанавливает соответствующий
     * менеджер безопасности и регистрирует MudServer под заданным именем
     * с помощью реестра RMI по умолчанию.
     **/
    public static void main(String[] args) {
     try {
      MudServer server;
      if (args.length == 1) {
       // Считываем состояние MUD из файла
       FileInputStream f = new FileInputStream(args[0]);
       ObjectInputStream in = new ObjectInputStream(new GZIPInputStream(f));
       server = (MudServer) in.readObject();
      }
      // Иначе создаем новую MUD с самого начала
      else server = new MudServer(args[0], args[1], args[2], args[3]);

      Naming.rebind(Mud.mudPrefix + server.mudname, server);
     }
     // При возникновении ошибки выводим сообщение.
     catch (Exception e) {
      System.out.println(e);
      System.out.println("Формат: java MudServer <savefile>\n" +
           " or: java MudServer <mudname> <password> " +
           "<placename> <description>");
      System.exit(1);
     }
    }

    /** Эта константа является номером версии и используется при сериализации */
    static final long serialVersionUID = 7453281245880199453L;
}
```

Класс MudPlace

В примере 16.5 показан класс `MudPlace`**, реализующий интерфейс** `RemoteMudPlace` **и действующий в качестве сервера для отдельной комнаты или места, находящегося внутри MUD. Это именно тот класс, который хранит описание комнаты и содержит списки находящихся в ней людей и предметов, а также выходов из нее. Это достаточно большой класс, но многие из определенных в нем удаленных методов имеют простые, а зачастую даже тривиальные реализации. Методы** `go()`**,** `createPlace()` **и** `linkTo()` **относятся к более сложным и интересным методам, они управляют сетевым обменом между объектами** `MudPlace`**.**

Обратите внимание, что класс `MudPlace` **— сериализуемый, поэтому** `MudPlace` **(и все связанные с ним комнаты) могут быть сериализованы вместе с** `MudServer`**, который ссылается на них. Однако поля** `names` **и** `people` **объявлены как** `transient`**, то есть они не сериализуются вместе с комнатой.**

Пример 16.5. MudPlace.java

```
package com.davidflanagan.examples.rmi;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.io.*;
import java.util.*;
import com.davidflanagan.examples.rmi.Mud.*;

/**
 * Этот класс реализует интерфейс RemoteMudPlace и экспортирует ряд
 * удаленных методов, которые являются центральной частью MUD.
 * MudClient взаимодействует в основном с этими методами.
 * Для предварительного знакомства см. комментарии к RemoteMudPlace.
 * Класс MudPlace является сериализуемым, поэтому комнаты могут
 * быть сохранены на диске вместе с содержащим их MUD-сервером.
 * Однако учтите, что поля name и people отмечены как временные (transient),
 * поэтому они не могут быть сериализованы вместе с их комнатой
 * (т. к. не имеет смысла пытаться сохранять объекты RemoteMudPerson,
 * даже если они могут быть сериализованы).
 **/
public class MudPlace extends UnicastRemoteObject
 implements RemoteMudPlace, Serializable
{
 String placename, description;          // информация о комнате
 Vector exits = new Vector();            // имена выходов из этой комнаты
 Vector destinations = new Vector();     // куда эти выходы ведут
 Vector things = new Vector();           // имена предметов в комнате
 Vector descriptions = new Vector();     // описания этих предметов
 transient Vector names = new Vector();  // имена людей в комнате
 transient Vector people = new Vector(); // объекты RemoteMudPerson
 MudServer server;                       // сервер для этой комнаты

 /** Конструктор без аргументов, служащий исключительно
     для целей десериализации. Не вызывайте его */
 public MudPlace() throws RemoteException { super(); }

 /**
  * Этот конструктор создает комнату и вызывает метод
  * сервера, регистрирующий этот объект, чтобы
  * он был доступен через свое имя.
  **/
 public MudPlace(MudServer server, String placename, String description)
  throws RemoteException, PlaceAlreadyExists
 {
  this.server = server;
```

```
 this.placename = placename;
 this.description = description;
 // Регистрируем комнату
 server.setPlaceName(this, placename);
}

/** Этот удаленный метод возвращает имя данной комнаты */
public String getPlaceName() throws RemoteException { return placename; }

/** Этот удаленный метод возвращает описание данной комнаты */
public String getDescription() throws RemoteException {
 return description;
}

/** Этот удаленный метод возвращает объект Vector, содержащий имена людей,
    находящихся в данной комнате */
public Vector getNames() throws RemoteException { return names; }

/** Этот удаленный метод возвращает объект Vector с именами
    предметов данной комнаты */
public Vector getThings() throws RemoteException { return things; }

/** Этот удаленный метод возвращает объект Vector с именами выходов
    из данной комнаты */
public Vector getExits() throws RemoteException { return exits; }

/**
 * Этот удаленный метод возвращает объект RemoteMudPerson,
 * соответствующий указанному имени, или генерирует
 * исключение, если такого человека здесь нет.
 **/
public RemoteMudPerson getPerson(String name)
 throws RemoteException, NoSuchPerson
{
 synchronized(names) {
  // Что если имеются 2 одинаковых имени?
  int i = names.indexOf(name);
  if (i == -1) throw new NoSuchPerson();
  return (RemoteMudPerson) people.elementAt(i);
 }
}

/**
 * Этот удаленный метод возвращает описание названного
 * предмета или генерирует исключение, если такого предмета
 * нет в этой комнате.
 **/
public String examineThing(String name) throws RemoteException, NoSuchThing
{
 synchronized(things) {
  int i = things.indexOf(name);
  if (i == -1) throw new NoSuchThing();
  return (String) descriptions.elementAt(i);
 }
```

```
}

/**
 * Этот удаленный метод перемещает указанный объект RemoteMudPerson
 * из этого места в заданном направлении (т. е. через заданный выход)
 * в какую-то другую находящуюся здесь комнату. Он генерирует исключение
 * в случаях, если заданный человек не находится в начальной комнате
 * или если он уже в комнате, в которую ведет выход, или данный выход
 * не существует, или он связан с комнатой, находящейся на другом
 * MUD-сервере, а этот сервер не функционирует.
 **/
public RemoteMudPlace go(RemoteMudPerson who, String direction)
 throws RemoteException, NotThere, AlreadyThere, NoSuchExit, LinkFailed
{
 // Проверяем, что направление правильное и, если это так,
 // получаем место назначения
 Object destination;
 synchronized(exits) {
  int i = exits.indexOf(direction);
  if (i == -1) throw new NoSuchExit();
  destination = destinations.elementAt(i);
 }

 // Если место назначения является строкой, оно представляет
 // комнату на другом сервере, поэтому соединяемся с ним.
 // В противном случае комната находится на этом сервере.
 // Генерируем исключение, если мы не можем соединиться с сервером.
 RemoteMudPlace newplace;
 if (destination instanceof String) {
  try {
   String t = (String) destination;
   int pos = t.indexOf('@');
   String url = t.substring(0, pos);
   String placename = t.substring(pos+1);
   RemoteMudServer s = (RemoteMudServer) Naming.lookup(url);
   newplace = s.getNamedPlace(placename);
  }
  catch (Exception e) { throw new LinkFailed(); }
 }
 // Если место назначения не является строкой,
 // тогда это объект типа RemoteMudPlace
 else newplace = (RemoteMudPlace) destination;

 // Проверяем, что человек находится здесь, и получаем его имя.
 // Генерируем исключение, если его здесь нет.
 String name = verifyPresence(who);

 // Убираем человека отсюда и сообщаем об этом всем оставшимся здесь.
 this.exit(who, name + " ушел в " + direction);

 // Помещаем человека в новое место. Посылаем сообщение
 // всем, кто уже находится в этой комнате.
 String fromwhere;
```

```
 if (newplace instanceof MudPlace) // идем в локальную комнату
  fromwhere = placename;
 else
  fromwhere = server.getMudName() + "." + placename;
 newplace.enter(who, name, name + "прибыл из: " + fromwhere);

 // Возвращаем клиенту новый объект RemoteMudPlace,
 // чтобы он знал, где мы сейчас находимся.
 return newplace;
}

/**
 * Этот удаленный метод посылает сообщение всем находящимся
 * в комнате. Используется для беседы с кем-либо.
 * Требует, чтобы говорящий находился в этой комнате.
 **/
public void speak(RemoteMudPerson speaker, String msg)
 throws RemoteException, NotThere
{
 String name = verifyPresence(speaker);
 tellEveryone(name + ":" + msg);
}

/**
 * Этот удаленный метод посылает сообщение всем находящимся в комнате.
 * Используется для выполнения действий, видимых людям.
 * Требует, чтобы человек, совершающий действие, находился в этой комнате.
 **/
public void act(RemoteMudPerson actor, String msg)
 throws RemoteException, NotThere
{
 String name = verifyPresence(actor);
 tellEveryone(name + " " + msg);
}

/**
 * Этот удаленный метод создает в комнате новый предмет.
 * Он требует, чтобы создающий его находился в данной комнате.
 **/
public void createThing(RemoteMudPerson creator,
      String name, String description)
 throws RemoteException, NotThere, AlreadyThere
{
 // Проверяем, здесь ли создающий
 String creatorname = verifyPresence(creator);
 synchronized(things) {
  // Проверяем, нет ли уже чего-нибудь с таким именем.
  if (things.indexOf(name) != -1) throw new AlreadyThere();
  // Вносим в соответствующие списки имя предмета и его описание
  things.addElement(name);
  descriptions.addElement(description);
 }
 // Сообщаем всем о новом предмете и о его создателе
```

```
 tellEveryone(creatorname + " создал " + name);
}

/**
 * Удаляем предмет из комнаты. Генерируем исключение,
 * если человек, удаляющий его, сам не находится в этой
 * комнате или в ней нет такого предмета.
 **/
public void destroyThing(RemoteMudPerson destroyer, String thing)
 throws RemoteException, NotThere, NoSuchThing
{
 // Проверяем, что человек, удаляющий предмет, здесь
 String name = verifyPresence(destroyer);
 synchronized(things) {
  // Проверяем, что в комнате есть предмет с таким именем
  int i = things.indexOf(thing);
  if (i == -1) throw new NoSuchThing();
  // и удаляем из списков его имя и описание
  things.removeElementAt(i);
  descriptions.removeElementAt(i);
 }
 // Сообщаем всем об удалении этого предмета.
 tellEveryone(name + " уничтожил " + thing);
}

/**
 * Создаем в этой MUD новую комнату с заданным именем и описанием.
 * В эту новую комнату из данного места можно попасть через указанный выход,
 * а это место доступно из новой комнаты через указанный вход.
 * Для того чтобы создать выход из этого места, создающий должен сам
 * находиться в нем.
 **/
public void createPlace(RemoteMudPerson creator,
      String exit, String entrance, String name,
      String description)
 throws RemoteException,NotThere,ExitAlreadyExists,PlaceAlreadyExists
{
 // Проверяем, что создающий на самом деле здесь
 String creatorname = verifyPresence(creator);
 // В данный момент времени выходы изменять может только один клиент
 synchronized(exits) {
  // Проверяем, что выход еще не существует.
  if (exits.indexOf(exit) != -1) throw new ExitAlreadyExists();
  // Создаем новую комнату, регистрируя ее имя на сервере
  MudPlace destination = new MudPlace(server, name, description);
  // Создаем связь оттуда сюда
  destination.exits.addElement(entrance);
  destination.destinations.addElement(this);
  // Создаем связь отсюда туда
  exits.addElement(exit);
  destinations.addElement(destination);
 }
```

```
 // Сообщаем всем о новом выходе и новой комнате за ним
 tellEveryone(creatorname + " создал новую комнату: " + exit);
}

/**
 * Создаем новый выход из этой MUD, связанный с указанным местом
 * в заданной MUD на заданном хосте (разумеется, он также может быть
 * использован для связи с определенной комнатой, находящейся в текущей MUD).
 * Из-за возможности возникновения взаимных блокировок этот метод
 * создает только связи типа "отсюда туда". Он не создает выходов
 * для возвращения оттуда сюда. Это должно выполняться
 * при помощи отдельного вызова.
 **/
public void linkTo(RemoteMudPerson linker, String exit,
     String hostname, String mudname, String placename)
 throws RemoteException, NotThere, ExitAlreadyExists, NoSuchPlace
{
 // Проверяем, что человек, создающий связь, действительно находится здесь
 String name = verifyPresence(linker);

 // Проверяем, что целевое место связи действительно существует.
 // Если нет - генерируем NoSuchPlace. Заметим, что NoSuchPlace
 // может также означать "NoSuchMud" (Нет такой MUD)
 // или "MudNotResponding" (MUD не отвечает).
 String url = "rmi://" + hostname + '/' + Mud.mudPrefix + mudname;
 try {
  RemoteMudServer s = (RemoteMudServer) Naming.lookup(url);
  RemoteMudPlace destination = s.getNamedPlace(placename);
 }
 catch (Exception e) { throw new NoSuchPlace(); }

 synchronized(exits) {
  // Проверяем, что выход еще не существует.
  if (exits.indexOf(exit) != -1) throw new ExitAlreadyExists();
  // Добавляем выход к списку имен выходов
  exits.addElement(exit);
  // Вносим место назначения в список мест назначения.
  // Заметим, что оно сохраняется в виде строки, а не в виде
  // объекта RemoteMudPlace. Это сделано из-за того, что когда удаленный
  // сервер останавливается, а затем снова запускается,
  // значение RemoteMudPlace становится неопределенным, тогда как строка
  // все еще действительна.
  destinations.addElement(url + '@' + placename);
 }
 // Сообщаем всем о новом выходе и о том, куда он ведет
 tellEveryone(name + " связал " + exit + " с " +
     "`" + placename + "` в MUD `" + mudname + "` на хосте " + hostname);
}

/**
 * Закрываем выход, ведущий из этой комнаты. Здесь не закрывается
 * обратный выход оттуда сюда. Обратите внимание, что данный метод
 * не уничтожает комнату, в которую ведет выход.
```

```
 * В текущей реализации нет способа удалить комнату.
 **/
public void close(RemoteMudPerson who, String exit)
 throws RemoteException, NotThere, NoSuchExit
{
 // Проверяем, что человек, закрывающий выход, действительно находится здесь
 String name = verifyPresence(who);
 synchronized(exits) {
  // Проверяем, что выход существует
  int i = exits.indexOf(exit);
  if (i == -1) throw new NoSuchExit();
  // Удаляем из списков его и его место назначения
  exits.removeElementAt(i);
  destinations.removeElementAt(i);
 }
 // Сообщаем всем, что выход больше не существует
 tellEveryone(name + " закрыл выход " + exit);
}

/**
 * Удаляем человека из этого места. Если есть сообщение, посылаем его всем,
 * кто остался в этом месте. Если указанного человека здесь нет, этот метод
 * ничего не делает и не генерирует каких-либо исключений.
 * Этот метод вызывается методом go(), и клиент должен вызывать
 * его при выходе пользователя. Вместе с тем клиент
 * не должен разрешать пользователю вызывать его напрямую.
 **/
public void exit(RemoteMudPerson who, String message)
 throws RemoteException
{
 String name;
 synchronized(names) {
  int i = people.indexOf(who);
  if (i == -1) return;
  names.removeElementAt(i);
  people.removeElementAt(i);
 }
 if (message != null) tellEveryone(message);
}

/**
 * Этот метод помещает человека в данное место, присваивая ему заданное имя
 * и отображая сообщения для всех остальных, находящихся в этом месте.
 * Этот метод вызывается из метода go(), и клиент должен вызывать его
 * для начальной комнаты, через которую человек входит в MUD.
 * Однако после того как этот человек попал в MUD, клиенту следует запретить
 * ему использовать метод go() и не разрешать пользователям напрямую вызывать
 * этот метод. При возникновении сетевых проблем клиент может вызвать этот
 * метод для восстановления человека в этом месте в случае, если он был
 * выброшен оттуда. (Человек будет выброшен из комнаты в случае, если сервер
 * попытается послать ему сообщение и получит исключение RemoteException.)
 **/
```

```
public void enter(RemoteMudPerson who, String name, String message)
 throws RemoteException, AlreadyThere
{
 // Посылаем сообщение всем, кто уже здесь.
 if (message != null) tellEveryone(message);

 // Добавляем человека в эту комнату.
 synchronized (names) {
  if (people.indexOf(who) != -1) throw new AlreadyThere();
  names.addElement(name);
  people.addElement(who);
 }
}

/**
 * Этот последний удаленный метод возвращает объект
 * сервера для MUD, в которой находится эта комната.
 * Клиент не должен разрешать пользователю вызывать этот метод.
 **/
public RemoteMudServer getServer() throws RemoteException {
 return server;
}

/**
 * Создаем и запускаем поток, рассылающий сообщения всем
 * находящимся в данном месте. Если, разговаривая с человеком,
 * он получает исключение RemoteException, он незаметно удаляет
 * этого человека из данного места. Он не является удаленным методом,
 * а используется локально несколькими удаленными методами.
 **/
protected void tellEveryone(final String message) {
 // Если здесь никого нет, не утруждаем себя отправкой сообщения!
 if (people.size() == 0) return;
 // Делаем копию списка находящихся здесь людей.
 // Сообщение посылается асинхронно, и список людей
 // может измениться, перед тем как сообщение достигнет каждого.
 final Vector recipients = (Vector) people.clone();
 // Создаем и запускаем поток для отправки сообщения, используя
 // безымянный класс. Мы делаем это из-за того, что отправка сообщения
 // всем находящимся в этом месте может занять некоторое время
 // (особенно на медленных и длинных сетях), а мы не хотим ждать.
 new Thread() {
   public void run() {
    // Цикл по всем получателям
    for(int i = 0; i < recipients.size(); i++) {
     RemoteMudPerson person =
      (RemoteMudPerson)recipients.elementAt(i);
     // Пробуем послать сообщение каждому.
     try { person.tell(message); }
     // При возникновении ошибки считаем, что это ошибка
     // сети или клиентского приложения, и незаметно
     // удаляем человека из этой комнаты.
```

```
      catch (RemoteException e) {
       try { MudPlace.this.exit(person, null); }
       catch (Exception ex) {}
      }
     }
    }
   }.start();
  }

  /**
   * Этот вспомогательный метод проверяет, здесь ли находится
   * указанный человек. Если здесь, он возвращает его имя.
   * Если же нет, он генерирует исключение NotThere
   **/
  protected String verifyPresence(RemoteMudPerson who) throws NotThere {
   int i = people.indexOf(who);
   if (i == -1) throw new NotThere();
   else return (String) names.elementAt(i);
  }

  /**
   * Этот метод используется для пользовательской десериализации.
   * Из-за того что векторы, хранящие людей и их имена, являются
   * непостоянными, они не сериализуются вместе с остальной частью
   * этой комнаты. Следовательно, при десериализации комнаты
   * эти векторы должны быть созданы заново (пустыми).
   **/
  private void readObject(ObjectInputStream in)
   throws IOException, ClassNotFoundException {
   // Считываем большую часть объекта, как обычно
   in.defaultReadObject();
   names = new Vector();  // Затем воссоздаем вектор имен
   people = new Vector(); // и вектор людей
  }

 /** Эта константа является номером версии и используется
      при сериализации */
  static final long serialVersionUID = 5090967989223703026L;
 }
```

Класс MudPerson

В примере 16.6 приведен класс `MudPerson`. Он является самым простым из всех удаленных объектов в системе MUD. Он реализует два удаленных метода, определенных в интерфейсе `RemoteMudPerson`, и кроме этого определяет несколько не удаленных методов, используемых классом `MudClient` из примера 16.7. Удаленные методы довольно просты: один из них просто возвращает вызывающему объекту строку описания, а другой записывает сообщение в поток, где его может увидеть пользователь.

Пример 16.6. MudPerson.java

```
package com.davidflanagan.examples.rmi;
import java.rmi.*;
import java.rmi.server.*;
import java.io.*;
import com.davidflanagan.examples.rmi.Mud.*;

/**
 * Это самый простой из всех удаленных объектов, реализуемых
 * нами для системы MUD. Он поддерживает только очень простое
 * состояние и имеет только два экспортируемых метода.
 **/
public class MudPerson extends UnicastRemoteObject implements
RemoteMudPerson {
 String name;            // Имя человека
 String description;     // Описание человека
 PrintWriter tellStream; // Куда посылать сообщения, которые мы принимаем

 public MudPerson(String n, String d, PrintWriter out)
  throws RemoteException
 {
  name = n;
  description = d;
  tellStream = out;
 }

 /** Возвращаем имя человека. Неудаленный метод */
 public String getName() { return name; }

 /** Задаем имя человека. Неудаленный метод */
 public void setName(String n) { name = n; }

 /** Устанавливаем описание человека. Неудаленный метод */
 public void setDescription(String d) { description = d; }

 /** Задаем поток, в который будут записываться предназначенные
     для нас сообщения. Не удаленный. */
 public void setTellStream(PrintWriter out) { tellStream = out; }

 /** Удаленный метод, возвращающий описание этого человека */
 public String getDescription() throws RemoteException {
  return description;
 }

 /**
  * Удаленный метод, доставляющий сообщение человеку.
  * То есть доставляет сообщение пользователю, управляющему "человеком"
  **/
 public void tell(String message) throws RemoteException {
  tellStream.println(message);
  tellStream.flush();
 }
}
```

Клиент MUD

Пример 16.7 является клиентской программой для системы MUD, разработанной в предыдущих примерах. Он использует метод `Naming.lookup()` для поиска объекта `RemoteMudServer`, который представляет указанную MUD на заданном хосте. Затем программа вызывает метод `getEntrance()` или `getNamedPlace()` этого объекта для получения начального объекта `MudPlace` и введения в него пользователя. После этого программа запрашивает у пользователя имя и описание для объекта `MudPerson`, который будет представлять его в системе MUD, создает объект `MudPerson` с этим именем и описанием, а затем помещает его в начальный объект `RemoteMudPlace`. Наконец, программа входит в цикл, в котором просит пользователя ввести команду и обрабатывает ее. Большинство команд, которые поддерживает этот клиент, просто вызывают один из удаленных методов объекта `RemoteMudPlace`, представляющего текущее положение пользователя в MUD. Конец командного цикла состоит из ряда секций `catch`, которые обрабатывают большое количество ошибочных ситуаций.

Перед началом использования класса `MudClient` вы должны запустить `MudServer`. Вы сможете сделать это с помощью следующих команд:

```
% cd com/davidflanagan/examples/rmi
% javac Mud*.java
% rmic -d ../../../../ com.davidflanagan.examples.rmi.MudServer
% rmic -d ../../../../ com.davidflanagan.examples.rmi.MudPlace
% rmic -d ../../../../ com.davidflanagan.examples.rmi.MudPerson
% rmiregistry &
% java com.davidflanagan.examples.rmi.MudServer MyMud muddy Lobby \
  'Большой мраморный холл с фикусами'
```

Запустив сервер с помощью этих команд, вы можете запустить клиент с помощью команды, подобной этой:

```
% java com.davidflanagan.examples.rmi.MudClient localhost MyMud
```

Пример 16.7. MudClient.java

```
package com.davidflanagan.examples.rmi;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.io.*;
import java.util.*;
import com.davidflanagan.examples.rmi.Mud.*;

/**
 * Этот класс является клиентской программой для MUD.
 * Метод main() устанавливает соединение с RemoteMudServer,
 * получает первоначальный объект RemoteMudPlace и создает
 * объект MudPerson, представляющий пользователя в MUD.
```

```
 * Затем он вызывает runMud() для помещения человека в это место
 * и начинает обработку команд пользователя. Методы getLine()
 * и getMultiLine() являются вспомогательными методами,
 * повсеместно используемыми для получения ввода пользователя.
 **/
public class MudClient {
 /**
 * Главная программа. Ожидает два или три аргумента:
 * 0) имя хоста, на котором запущен MUD-сервер
 * 1) имя MUD на этом сервере
 * 2) имя стартовой комнаты внутри этой (необязательный аргумент).
 *
 * Использует method Naming.lookup() для получения объекта
 * RemoteMudServer для указанной MUD, находящейся на указанном хосте.
 * Затем она использует метод getEntrance() или getNamedPlace()
 * объекта RemoteMudServer для получения объекта RemoteMudPlace.
 * Она запрашивает у пользователя его имя и описание и создает
 * объект MudPerson. Наконец, она передает этого человека и комнату
 * в метод runMud() для начала взаимодействия с MUD.
 **/
 public static void main(String[] args) {
 try {
  // Каждая MUD уникально идентифицируется хостом и своим именем.
  String hostname = args[0];
  String mudname = args[1];
  // Каждая комната в MUD имеет уникальное имя
  String placename = null;
  if (args.length > 2) placename = args[2];

  // Ищем объект RemoteMudServer для указанной MUD с помощью
  // реестра по умолчанию на заданном хосте. Обратите внимание
  // на использование константы Mud.mudPrefix с целью
  // предотвращения конфликта имен в реестре.
  RemoteMudServer server =
  (RemoteMudServer)Naming.lookup("rmi://" + hostname + "/" +
      Mud.mudPrefix + mudname);

  // Если пользователь не указал комнату в MUD, используем
  // getEntrance() для получения начальной комнаты.
  // Иначе вызываем getNamedPlace() для поиска заданной комнаты.
  RemoteMudPlace location = null;
  if (placename == null) location = server.getEntrance();
  else location = (RemoteMudPlace) server.getNamedPlace(placename);

  // Приветствуем пользователя и запрашиваем его имя и описание.
  // Используем для этого определенные ниже методы
  // getLine() и getMultiLine().
  System.out.println("Добро пожаловать в " + mudname);
  String name = getLine("Введите ваше имя: ");
  String description = getMultiLine("Пожалуйста, опишите, " +
      "что видят люди, глядя на Вас:");

  // Определяем поток вывода, используемый объектом MudPerson
```

```
 // для вывода сообщений, посылаемых пользователю.
 // Мы будем использовать консоль.
 PrintWriter myout = new PrintWriter(System.out);

 // Создаем объект MudPerson для представления пользователя в MUD.
 // Используем заданное имя,описание и поток вывода.
 MudPerson me = new MudPerson(name, description, myout);

 // Уменьшаем приотитет этого потока на один уровень
 // для того, чтобы широковещательные сообщения могли
 // появляться даже в том случае, когда мы блокированы
 // операцией ввода/вывода. Это необходимо на платформе
 // Linux, но может быть необязательным на всех остальных платформах
 int pri = Thread.currentThread().getPriority();
 Thread.currentThread().setPriority(pri-1);

 // Наконец, помещаем MudPerson в RemoteMudPlace и начинаем
 // запрашивать команды у пользователя.
 runMud(location, me);
}
// При возникновении ошибки выводим сообщение и выходим.
catch (Exception e) {
 System.out.println(e);
 System.out.println("Формат: java MudClient <host> <mud> [<place>]");
 System.exit(1);
}
}

/**
* Этот метод является главным циклом программы MudClient.
* Он помещает человека в комнату (с помощью метода enter()
* объекта RemoteMudPlace). Затем он вызывает метод look()
* для вывода описания комнаты пользователю и входит
* в командный цикл для запроса у пользователя команд и их обработки
**/
public static void runMud(RemoteMudPlace entrance, MudPerson me)
throws RemoteException
{
RemoteMudPlace location = entrance; // Текущая комната
String myname = me.getName();       // Имя человека
String placename = null;            // Имя текущей комнаты
String mudname = null;              // Имя MUD, содержащей эту комнату

try {
 // Входим в MUD
 location.enter(me, myname, myname + " вошел в игру.");
 // Выясняем, где мы находимся (для подсказки)
 mudname = location.getServer().getMudName();
 placename = location.getPlaceName();
 // Описываем место пользователю
 look(location);
}
catch (Exception e) {
```

```
 System.out.println(e);
 System.exit(1);
}

// Сейчас, когда мы вошли в MUD, начинаем командный цикл
// обработки команд пользователя. Заметим, что внизу
// цикла находится массивный блок операторов catch,
// предназначенный для обработки всех ошибок, которые
// могут случиться при каждом проходе цикла.
for(;;) { // Цикл, пока пользователь не ввел "quit"
 try { // Перехватываем все исключения, имеющие место в цикле
 // Делаем небольшую паузу перед выводом приглашения,
 // чтобы дать возможность появиться выводу, косвенно
 // сгенерированному последней командой.
 try { Thread.sleep(200); } catch (InterruptedException e) {}

 // Выводим приглашение и получаем строку от пользователя
 String line = getLine(mudname + '.' + placename + "> ");

 // Разбиваем строку на команду и агрумент, содержащий
 // остаток строки. Приводим команду к нижнему регистру.
 String cmd, arg;
 int i = line.indexOf(' ');
 if (i == -1) { cmd = line; arg = null; }
 else {
  cmd = line.substring(0, i).toLowerCase();
  arg = line.substring(i+1);
 }
 if (arg == null) arg = "";

 // Начинаем обрабатывать команду. Далее следует громадный
 // повторяющийся оператор if/else, преобразующий все
 // команды, поддерживаемые этим клиентом. Многие из этих
 // команд просто вызывают один из удаленных методов
 // текущего объекта RemoteMudPlace. Некоторым придется
 // выполнить небольшую дополнительную обработку.

 // LOOK: Описывает комнату, ее предметы, людей и выходы
 if (cmd.equals("look")) look(location);
 // EXAMINE: Описывает указанный предмет
 else if (cmd.equals("examine"))
  System.out.println(location.examineThing(arg));
 // DESCRIBE: Описывает указанного человека
 else if (cmd.equals("describe")) {
  try {
  RemoteMudPerson p = location.getPerson(arg);
  System.out.println(p.getDescription());
  }
  catch(RemoteException e) {
  System.out.println("У " + arg + " техническая " +
     "проблема. Нет доступного описания");
  }
 }
```

```
// GO: Идем в указанном направлении
else if (cmd.equals("go")) {
 location = location.go(me, arg);
 mudname = location.getServer().getMudName();
 placename = location.getPlaceName();
 look(location);
}
// SAY: Говорим всем что-либо
else if (cmd.equals("say")) location.speak(me, arg);
// DO: Делаем что-нибудь, что будет описано всем
else if (cmd.equals("do")) location.act(me, arg);
// TALK: Говорим что-нибудь одному заданному человеку
else if (cmd.equals("talk")) {
 try {
 RemoteMudPerson p = location.getPerson(arg);
 String msg = getLine("Что вы хотите сказать?: ");
 p.tell(myname + " сказал \"" + msg + "\"");
 }
 catch (RemoteException e) {
 System.out.println("У " + arg + " техническая " +
    "проблема. Не могу говорить с ними.");
 }
}
// CHANGE: Меняем собственное описание
else if (cmd.equals("change"))
 me.setDescription(
  getMultiLine("Опишите себя для других: "));
// CREATE: Создаем в этом месте новый предмет
else if (cmd.equals("create")) {
 if (arg.length() == 0)
 throw new IllegalArgumentException("ожидалось имя");
 String desc = getMultiLine("Пожалуйста, опишите " +
    arg + ": ");
 location.createThing(me, arg, desc);
}
// DESTROY: Уничтожаем указанный предмет
else if (cmd.equals("destroy")) location.destroyThing(me, arg);
// OPEN: Создаем новую комнату и соединяем с ней это
// место через выход, указанный в качестве параметра.
else if (cmd.equals("open")) {
 if (arg.length() == 0)
 throw new IllegalArgumentException("ожидалось направление");
 String name = getLine("Какое имя у той комнаты?: ");
 String back = getLine("Какое направление " +
    "оттуда сюда?: ");
 String desc = getMultiLine("Пожалуйста, опишите " +
    name + ":");
 location.createPlace(me, arg, back, name, desc);
}
// CLOSE: Закрываем указанный выход. Заметьте: выход
// закрываем только в одном направлении и не уничтожаем комнату.
```

```
else if (cmd.equals("close")) {
 if (arg.length() == 0)
 throw new IllegalArgumentException("ожидалось направление");
 location.close(me, arg);
}
// LINK: Создаем новый выход, соединенный с существующей
// комнатой, которая может находиться в другой MUD,
// запущенной на другом хосте
else if (cmd.equals("link")) {
 if (arg.length() == 0)
 throw new IllegalArgumentException("ожидалось направление");
 String host = getLine("С каким хостом вы соединяетесь?: ");
 String mud = getLine("Какое имя MUD на этом хосте?: ");
 String place = getLine("Какое имя комнаты в той MUD?: ");
 location.linkTo(me, arg, host, mud, place);
 System.out.println("Не забудьте создать связь " + "оттуда сюда!");
}
// DUMP: Сохраняем состояние MUD в указанном файле,
// если указан правильный пароль
else if (cmd.equals("dump")) {
 if (arg.length() == 0)
 throw new IllegalArgumentException("ожидалось имя файла");
 String password = getLine("Пароль: ");
 location.getServer().dump(password, arg);
}
// QUIT: Выход из игры
else if (cmd.equals("quit")) {
 try { location.exit(me, myname + " вышел."); }
 catch (Exception e) {}
 System.out.println("Пока.");
 System.out.flush();
 System.exit(0);
}
// HELP: Распечатываем большое справочное сообщение
else if (cmd.equals("help")) System.out.println(help);
// Иначе это нераспознанная команда.
else System.out.println("Неизвестная команда. Введите 'help'.");
}
// Обрабатываем многие возможные типы MudException
catch (MudException e) {
if (e instanceof NoSuchThing)
 System.out.println("Здесь нет такого предмета.");
else if (e instanceof NoSuchPerson)
 System.out.println("Здесь нет никого с таким именем.");
else if (e instanceof NoSuchExit)
 System.out.println("В этом направлении нет выхода.");
else if (e instanceof NoSuchPlace)
 System.out.println("Нет такой комнаты.");
else if (e instanceof ExitAlreadyExists)
 System.out.println("В этом направлении " +
   "выход уже существует.");
```

```
 else if (e instanceof PlaceAlreadyExists)
  System.out.println("Комната с таким именем " +
    "уже существует.");
 else if (e instanceof LinkFailed)
  System.out.println("Этот выход не функционирует.");
 else if (e instanceof BadPassword)
  System.out.println("Неверный пароль.");
 else if (e instanceof NotThere) // Не должно случаться
  System.out.println("Вы не можете это сделать, " +
    "если Вы находитесь не там.");
 else if (e instanceof AlreadyThere) // Не должно случаться
  System.out.println("Вы не можете уйти туда; " +
    "Вы уже там.");
 }
 // Обрабатываем исключения RMI
 catch (RemoteException e) {
 System.out.println("У MUD технические проблемы.");
 System.out.println("Возможно, сервер потерпел аварию:");
 System.out.println(e);
 }
 // Обрабатываем другие ошибочные ситуации.
 catch (Exception e) {
 System.out.println("Синтаксическая или другая ошибка:");
 System.out.println(e);
 System.out.println("Используйте команду 'help'.");
 }
}
}

/**
* Этот вспомогательный метод используется несколько раз
* в приведенном выше методе runMud(). Он выводит имя
* и описание текущей комнаты (включая имя MUD, содержащей
* эту комнату), а также показывает список предметов,
* людей и выходов, имеющихся в текущей комнате.
**/
public static void look(RemoteMudPlace p)
throws RemoteException, MudException
{
String mudname = p.getServer().getMudName(); // Имя Mud
String placename = p.getPlaceName();         // Имя комнаты
String description = p.getDescription();     // Описание комнаты
Vector things = p.getThings();               // Список предметов
Vector names = p.getNames();                 // Список людей
Vector exits = p.getExits();                 // Список выходов отсюда

// Распечатываем их все
System.out.println("Вы находитесь в " + placename +
   " входящем в Mud: " + mudname);
System.out.println(description);
System.out.print("\nПредметы: ");
for(int i = 0; i < things.size(); i++) { // Выводим список предметов
```

```
 if (i > 0) System.out.print(", ");
 System.out.print(things.elementAt(i));
}
System.out.print("\nЛюди: ");
for(int i = 0; i < names.size(); i++) { // Выводим список людей
 if (i > 0) System.out.print(", ");
 System.out.print(names.elementAt(i));
}
System.out.print("\nВыходы: ");
for(int i = 0; i < exits.size(); i++) { // Выводим список выходов
 if (i > 0) System.out.print(", ");
 System.out.print(exits.elementAt(i));
}
System.out.println();    // Пустая строка
System.out.flush();      // Показываем его сейчас!
}

/** Этот статический входной поток читает строки с консоли */
static BufferedReader in =
new BufferedReader(new InputStreamReader(System.in));

/**
* Вспомогательный метод для вывода приглашения пользователю
* и получения строки ввода. Он гарантирует, что строка
* не будет пустой, и обрезает все пробелы в начале и конце строки.
**/
public static String getLine(String prompt) {
String line = null;
do {   // Цикл, пока не будет введена непустая строка
 try {
 System.out.print(prompt);  // Выводим приглашение
 System.out.flush();        // Выводим прямо сейчас
 line = in.readLine();      // Получаем строку ввода
 if (line != null) line = line.trim(); // Обрезаем пробелы
 } catch (Exception e) {}   // Игнорируем все ошибки
} while((line == null) || (line.length() == 0));
return line;
}

/**
* Вспомогательный метод для получения от пользователя
* многострочного ввода. Он выводит приглашение для ввода,
* показывет инструкции и гарантирует, что ввод не будет
* пустым. Кроме этого он разрешает пользователю ввести
* имя файла, из которого будет считан текст.
**/
public static String getMultiLine(String prompt) {
String text = "";
// Мы прервем этот цикл, когда получим непустой ввод
for(;;) {
 try {
 BufferedReader br = in;     // Поток для чтения
```

```
  System.out.println(prompt); // Выводим приглашение
  // Показываем инструкции
  System.out.println("Вы можете ввести несколько строк. " +
     "Завершайте ввод строкой, содержащей только'.'.\n" +
     "Или введите '<<', а затем имя файла");
  // Делаем приглашение и инструкции видимыми.
  System.out.flush();
  // Считываем строки
  String line;
  while((line = br.readLine()) != null) { // До EOF
   if (line.equals(".")) break; // Или до точки
   // Или, если задан файл, начинаем читать из него вместо консоли.
   if (line.trim().startsWith("<<")) {
   String filename = line.trim().substring(2).trim();
   br = new BufferedReader(new FileReader(filename));
   continue; // Не считаем << частью ввода
   }
   // Добавляем строку к формируемому вводу
   else text += line + "\n";
  }
  // Если мы получили хотя бы одну строку, возвращаем ее.
  // Иначе отчитываем пользователя и возвращаемся назад
  // к приглашению и инструкциям.
  if (text.length() > 0) return text;
  else System.out.println("Пожалуйста, введите хотя бы одну строку.");
  }
  // Если были ошибки, например ошибка ввода/вывода
  // при чтении из файла, показываем ошибку и начинаем цикл
  // сначала, выводя приглашение и инструкции
  catch(Exception e) { System.out.println(e); }
 }
 }

 /** Это строка формата, которая объясняет доступные команды */
 static final String help =
 "Команды:\n" +
 "look: осмотреться\n" +
 "examine <предмет>: детально изучить указанный предмет\n" +
 "describe <человек>: описать указанного человека\n" +
 "go <направление>: идти в указанном направлении (т. е. через указанный
выход)\n" +
 "say <сообщение>: говорить что-то всем\n" +
 "do <сообщение>: сказать всем, что вы что-то сделали\n" +
 "talk <человек>: говорить одному человеку. Будет запрошено сообщение\n" +
 "change: сменить ваше описание. Будет запрошен ввод описания\n" +
 "create <предмет>: создать новый предмет. Запрашивается описание\n" +
 "destroy <предмет>: уничтожить предмет\n" +
 "open <направление>: создать смежную комнату. Запрашивается ввод\n"+
 "close <направление>: закрыть выход из этого места\n" +
 "link <направление>: создать выход в существующее место,\n" +
 "      возможно, на другом сервере. Будет запрошен ввод.\n" +
```

```
        "dump <имя_файла>: сохранить состояние сервера. Запрашивается пароль\n" +
        "quit: выйти из Mud\n" +
        "help: показать это сообщение";
    }
```

Расширенный RMI

В RMI есть несколько расширенных возможностей, рассмотрение которых выходит за рамки этой книги, но о которых нужно знать. Здесь я только кратко опишу эти возможности. Подробную информацию можно найти в книге «Java Enterprise in a Nutshell».

Удаленная загрузка классов

В идеале клиенту удаленного объекта нужен только прямой доступ к удаленному интерфейсу; ему не требуется знать ничего относительно реализации этого интерфейса. Однако в примерах, которые мы видели в этой главе, клиенту также требуется доступ к реализации класса заглушки. (А на практике вы, вероятно, запускали и клиент, и сервер на одной машине с одинаковым путем к классам, поэтому они совместно использовали все классы.)

Необходимость распространять заглушки реализации вместе с клиентскими программами может составить дополнительные сложности, особенно когда реализация сервера постоянно меняется. К счастью, RMI предоставляет механизм, позволяющий клиенту дистанционно загружать с сетевого сервера необходимые ему классы заглушки. К сожалению, выполнение этих действий требует некоторых усилий. Во-первых, в вашем распоряжении должен быть работающий веб-сервер, способный предоставить для загрузки классы заглушкек. Во-вторых, вы должны установить на всех ваших клиентах менеджер безопасности (secutity manager) с целью защиты от злонамеренного кода, возможно, содержащегося в загруженных классах заглушек (и помните, что RMI-сервер, использующий другие удаленные объекты, сам является RMI-клиентом). В-третьих, поскольку вы установили менеджер безопасности, вы должна явно указать политики безопасности, которые позволят вашим RMI-клиентам устанавливать сетевые соединения, требуемые им для связи с удаленными объектами. За дополнительной информацией обращайтесь к книге «Java Enterprise in a Nutshell».

Активация

В примерах этой главы, посвященных RMI, процесс, реализующий удаленный объект, должен выполняться постоянно в ожидании подключений клиентов. Начиная с Java 1.2 служба активации RMI дает вам возможность создавать удаленные объекты, которые не выполняются все время, а создаются и активируются только при необходимос-

ти. Активация также разрешает использование постоянных удаленных ссылок, которые могут оставаться действительными даже при сбоях сервера.

Для того чтобы реализовать активируемый удаленный объект, вам вместо используемого в этой главе класса `UnicastRemoteObject` нужно расширять класс `java.rmi.activation.Activatable`. Создавая класс, производный от `Activatable`, вы должны определить конструктор инициализации, указывающий местоположение, с которого может быть загружен класс. Помимо этого вы должны определить конструктор активации (с другими параметрами), используемый службой активации для активации вашего объекта, когда тот будет запрошен клиентом.

После того как вы реализуете ваш активируемый удаленный объект, вы можете создать его начальный экземпляр и зарегистрировать его в службе активации или можете создать объект `ActivationDesc`, который сообщит службе активации, как создать его, когда он потребуется. В любом случае сама служба активации должна быть запущена. Вы можете запустить эту службу с помощью команды *rmid*, распространяемой вместе с Java SDK в версиях Java 1.2 и выше. Очень удобно то, что *rmid* также может выполнять и функцию *rmiregistry*.

Все, что было описано здесь, является деталями серверной реализации: клиент не может увидеть разницы между активируемыми и обычными удаленными объектами. В этом разделе был дан только краткий обзор создания активируемых объектов, для получения дополнительной информации см. «Java Enterprise in a Nutshell».

Совместимость CORBA и RMI/IIOP

Одной из слабых сторон традиционных удаленных объектов RMI является то, что они работают только в том случае, когда для реализации как клиента, так и сервера используется Java. С другой стороны – это достоинство, позволяющее сохранить инфраструктуру удаленных методов простой и легкой в использовании. Новая технология с названием RMI-IIOP позволяет вам использовать удаленные объекты RMI с помощью сетевого протокола IIOP. IIOP расшифровывается как Internet Inter-ORB Protocol: протокол, используемый стандартом распределенных объектов CORBA. RMI-IIOP реализуется пакетом `javax.rmi` и является стандартной частью Java версии 1.3 и выше.

Удаленные объекты RMI не могут автоматически использовать протокол IIOP: вы должны реализовывать их специально, путем наследования от класса `javax.rmi.PortableRemoteObject` и выполнения ряда других шагов. Хотя он и требует некоторых дополнительных усилий, использование RMI-IIOP может иметь большую ценность, если вы работаете в разнородном окружении и хотите связать удаленные объекты Java с существующими удаленными объектами, реализованными с применением стандарта CORBA. Для краткого обзора см. «Java Enter-

prise in a Nutshell», а также *http://java.sun.com/products/rmi-iiop/* и документацию по RMI-IIOP в Java SDK для полной информации.

Упражнения

16-1. Доработайте пример удаленного банковского обслуживания из этой главы так, чтобы клиентам банка разрешалось брать в банке ссуду на уровне некоторого максимального кредита, а также использовать деньги с их счета для оплаты своего долга. Добавьте к интерфейсу `RemoteBank` методы `borrow()` (занять) и `repay()` (возместить), реализуйте эти методы на сервере и измените клиент так, чтобы он мог вызывать эти методы, когда они понадобятся пользователю.

16-2. Программа *rmiregistry* предоставляет для RMI-приложений простую службу имен. Она позволяет серверам регистрировать имена удаленных объектов, которые они обслуживают, а клиентам позволяет искать эти удаленные объекты по имени. Так как эта служба является глобальным реестром, доступным всем удаленным службам, есть вероятность конфликта имен. По этой причине, если какой-либо службе требуется определить имена для серии удаленных объектов, как правило, она должна организовать собственный реестр. Таким образом, клиент может использовать глобальный реестр для поиска реестра системы имен определенной службы, а затем использовать этот реестр для поиска собственных объектов этой службы.

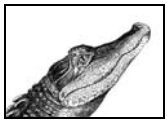
Напишите сервер с использованием RMI, который предоставлял бы такую пользовательскую службу имен. Он должен экспортировать удаленные методы, соответствующие методам `bind()`, `rebind()`, `unbind()` и `lookup()` класса `Naming`. Возможно, вы захотите использовать объект `java.util.Map` для связывания имен с удаленными объектами.

16-3. В примере MUD из этой главы для представления комнат, или мест, находящихся в MUD, и людей, взаимодействующих в MUD, используются удаленные объекты. Однако предметы, которые появляются в MUD, не являются удаленными объектами. Они представляют собой просто часть состояния каждой комнаты в MUD.

Доработайте этот пример так, чтобы предметы были настоящими удаленными объектами. Определите интерфейс `MudThing`, который расширял бы `Remote`. Он должен иметь метод `getDescription()`, возвращающий описание предмета. Измените интерфейс `MudPlace` и класс `RemoteMudPlace` для введения в них методов, которые позволили бы добавлять и удалять объекты `MudThing` из комнаты.

Создайте простейшую реализацию `MudThing`, которая просто возвращала бы из своего метода `getDescription()` статическую строку. Затем определите другую реализацию `MudThing` под именем `Clock`. Этот класс должен иметь более активное поведение: каждый раз при вызове его метода `getDescription()` он должен возвращать строку, отображающую текущее время. Измените сервер MUD так, чтобы он размещал объект `Clock` на входе в MUD.

16-4. Еще раз модифицируйте пример MUD так, чтобы объекты `MudPerson` могли подбирать объекты `MudThing`, которые они находят в `MudPlace`, носить их с собой, оставлять их в других местах и давать другим людям. Реализуйте, по крайней мере, три новых метода: `pickup()`, `drop()` и `give()`. Измените клиент MUD так, чтобы он поддерживал команды `pickup` (подбирать), `drop` (оставлять) и `give` (отдавать).



Глава 17

Доступ к базам данных при помощи SQL

В этой главе рассказывается, как вы можете взаимодействовать с сервером базы данных при помощи программного интерфейса JDBC, содержащегося в пакете `java.sql`. JDBC является интерфейсом, позволяющим Java-программам взаимодействовать с сервером базы данных с использованием команд языка SQL (Structured Query Language, структурированный язык запросов). Обратите внимание, что JDBC является программным интерфейсом для SQL, а не встроенным SQL-механизмом для Java.

Пакет `java.sql` предоставляет достаточно простой механизм для отправки SQL-запросов базе данных и для приема результатов выполнения этих запросов. Таким образом, если предположить, что у вас уже есть определенный опыт работы с базами данных и SQL, эта глава не будет представлять для вас большой сложности. И наоборот, если вы не работали с базами данных раньше, то для того чтобы извлечь пользу из примеров этой главы и JDBC в целом, вам потребуется изучить основы SQL-синтаксиса и некоторые общие принципы программирования баз данных. Перед тем как продолжить, я попытаюсь объяснить некоторые основные понятия, чтобы вы могли представить себе, что можно сделать с помощью JDBC, но полное изложение темы программирования баз данных выходит за рамки данной главы. Книга «Java Enterprise in a Nutshell» содержит более основательное введение в JDBC, справочник по SQL и программному интерфейсу для пакета `java.sql`.

Для того чтобы запустить примеры, приведенные в этой главе, вы должны получить доступ к базе данных, а также получить и установить соответствующий ей драйвер JDBC. Если у вас еще нет сервера базы данных для работы, вы можете воспользоваться одной из доступ-

ных сегодня превосходных баз данных с открытым исходным кодом. Примеры этой главы были протестированы с MySQL, открытой базой данных, доступной на сайте *http://www.mysql.com*, и с PostgreSQL, еще одним открытым сервером, доступным на *http://www.postgresql.org*. JDBC-драйверы для этих серверов баз данных могут быть загружены с этих же сайтов. Учтите, что серверы баз данных представляют собой сложные программные продукты. Загрузка и установка базы данных может потребовать значительных усилий.

Когда вы определитесь, какой сервер баз данных использовать, изучите прилагаемую к нему документацию. Перед тем как запускать примеры этой главы, вы должны изучить процедуру администрирования сервера. В частности, вам нужно будет узнать, как можно создать тестовую базу данных (или попросить администратора баз данных создать ее для вас). Если вы новичок в области баз данных, возможно, разобраться со всем этим будет для вас даже сложнее, чем изучить процесс программирования с помощью JDBC.

Доступ к базе данных

В примере 17.1 приведена программа, которая соединяется с базой данных, а затем в цикле запрашивает у пользователя SQL-оператор, передает его базе данных и выводит результат. В нем демонстрируются четыре важнейшие технологии программирования с использованием JDBC: регистрация драйвера для базы данных, использование класса `DriverManager` для получения объекта `Connection`, представляющего соединение с базой данных, отправка SQL-оператора базе данных с помощью объекта `Statement` и прием результатов запроса с помощью объекта `ResultSet`. Перед тем как перейти к программе `ExecuteSQL`, давайте рассмотрим эти базовые технологии.

Одна из интересных особенностей пакета `java.sql` состоит в том, что самые важные его члены, такие как `Connection`, `Statement` и `ResultSet`, являются интерфейсами, а не классами. Основная задача JDBC – скрыть детали, связанные с доступом к отдельным типам систем баз данных, и использование данных интерфейсов делает это возможным. JDBC-драйвер является набором классов, реализующих эти интерфейсы для конкретной системы баз данных. Разные системы баз данных требуют разных драйверов. Как разработчику приложений вам не нужно беспокоиться о деталях реализации этих базовых классов. Все, что вам нужно, – это написать код, вызывающий методы, определенные в различных интерфейсах.

Класс `DriverManager` обеспечивает управление всеми находящимися в системе JDBC-драйверами. Поэтому первым делом JDBC-программа должна зарегистрировать драйвер, соответствующий типу используемой базы данных. Согласно принятому соглашению классы JDBC-драйверов регистрируют себя сами при помощи `DriverManager` во время

своей первой загрузки. Поэтому на практике все, что вам нужно сделать, – это загрузить класс драйвера, позволив ему зарегистрировать себя. Одним из самых простых способов сделать это является вызов метода `Class.forName()`. Данный метод принимает аргумент типа `String`, содержащий имя класса, поэтому можно просто передать имя драйвера в командной строке вместо жесткого кодирования класса драйвера в вашей программе. Обратите внимание, что на этом шаге происходит просто загрузка драйвера и регистрация его в объекте `DriverManager`. Здесь еще не указывается, что программа действительно использует драйвер. Если программе необходимо использовать несколько баз данных, она может загрузить на этом шаге несколько классов драйверов. Затем наступает этап выбора драйвера, на котором программа реально соединяется с базой данных.

После того как необходимый драйвер загружен (и зарегистрирован), JDBC-программа может соединиться с базой данных с помощью вызова метода `DriverManager.getConnection()`. Вы указываете базу данных для соединения при помощи строки *jdbc*: URL, где URL имеет следующий синтаксис:

```
jdbc:subprotocol://host:port/databasename
```

Элемент *subprotocol* в этом URL задает тип системы используемой базы данных. Класс `DriverManager` использует эту часть URL для выбора подходящего драйвера из списка зарегистрированных драйверов. Если `DriverManager` не может найти JDBC-драйвер для данной базы данных, он генерирует исключение `SQLException`.

При разработке примеров этой главы я пользовался базой данных MySQL, поэтому для установления соединения с базой данных мне приходилось использовать URL наподобие следующего:

```
jdbc:mysql://dbserver.mydomain.com:1234/mydb
```

Этот URL указывает, что JDBC должен соединиться с базой данных «mydb», хранящейся в сервере баз данных MySQL, работающем на хосте *dbserver.mydomain.com* и прослушивающем порт 1234.

Если вы запускаете сервер баз данных на том же хосте, на котором запущена ваша Java-программа, вы можете опустить часть URL, содержащую имя хоста. Если ваш сервер баз данных прослушивает порт с адресом, принятым по умолчанию (как обычно и бывает), вы можете не указывать номер порта. Здесь приведен еще один URL, работающий с сервером PostgeSQL:

```
jdbc:postgresql://dbserver.mydomain.com/mydb
```

Или, если вы соединяетесь с сервером, работающим на локальном хосте:

```
jdbc:postgresql:mydb
```

Метод DriverManager.getConnection() возвращает объект, реализующий интерфейс Connection. Этот объект представляет соединение с базой данных, и вы можете использовать его для взаимодействия с базой. Метод createStatement() объекта Connection создает объект, реализующий интерфейс Statement, являющийся тем, чем вы пользуетесь при отправке SQL-запросов и обновлении базы данных. Методы executeQuery() и executeUpdate() объекта Statement служат для отправки соответственно запросов и обновлений, тогда как многоцелевой метод execute() используется для отправки команд, которые могут быть как запросами, так и командами обновлений.

После того как вы отправили базе данных запрос, используйте метод getResultSet() для получения объекта, реализующего интерфейс ResultSet. Этот объект представляет данные, возвращенные SQL-запросом. Они упорядочены в строки и столбцы наподобие таблицы. Объект ResultSet возвращает свои данные по одной строке за раз. Для перемещения от текущей строки к следующей вы можете использовать метод next(). ResultSet предоставляет большое количество методов getX(), позволяющих вам извлечь данные из каждого столбца текущей строки в виде чисел различных типов.

В Java 1.0 программный интерфейс JDBC был обновлен до версии 2.0. В JDBC 2.0 могут быть созданы прокручиваемые (scrollable) итоговые наборы значений. Это означает, что кроме метода next() вы можете использовать метод previous() – для перемещения к предыдущей строке, методы first() и last() – для перемещения соответственно к первой и последней строкам и методы absolute() и relative() – для перемещения к любой строке, заданной абсолютным или относительным номером строки. Если ваш сервер баз данных поддерживает прокручиваемые итоговые наборы и вы используете драйвер, совместимый с JDBC 2.0, то при создании объекта Statement вы можете задать прокручиваемый итоговый набор. Для того чтобы сделать это, воспользуйтесь версией метода cteateStatement(), принимающей два параметра, как это продемонстрировано в следующем коде:

```
Statement s = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                         ResultSet.CONCUR_READ_ONLY);
```

Для двух аргументов метода createStatement() есть и другие допустимые значения, но их рассмотрение выходит за рамки этой главы. Дополнительную информацию см. в книге «Java Enterprise in a Nutshell».

Теперь, когда вы имеете представление о базовых технологиях, используемых в JDBC-программах, давайте перейдем к примеру 17.1. В программе ExecuteSQL используются все только что рассмотренные технологии соединения с базой данных, выполнения SQL-операторов и отображения результатов. Для определения имени класса JDBC-драйвера, URL базы данных и других параметров, необходимых для соединения с базой данных, программа выполняет анализ аргументов своей команд-

ной строки. Например, вы могли бы вызвать программу ExecuteSQL и ввести простой запрос наподобие следующего (заметим, что эта длинная команда Java разбита здесь на две строки):

```
% java com.davidflanagan.examples.sql.ExecuteSQL -d org.gjt.mm.mysql.Driver \
       -u java -p nut jdbc:mysql://db.domain.com/api
sql> SELECT * FROM package WHERE name LIKE '%.rmi%'
+----+--------------------------------+
| id |              name              |
+----+--------------------------------+
| 14 | java.rmi                       |
| 15 | java.rmi.dgc                   |
| 16 | java.rmi.registry              |
| 17 | java.rmi.server                |
+----+--------------------------------+
sql> quit
```

Обратите внимание, что для выполнения SQL-операторов в программе ExecuteSQL используется метод execute() объекта Statement. Поскольку пользователь может ввести любой тип SQL-оператора, вы должны применять общецелевой метод. Если execute() возвращает true, то SQL-оператор был запросом, поэтому программа получает объект ResultSet и выводит результат запроса. В противном случае оператор являлся командой обновления, поэтому программа просто выводит информацию о том, на какое количество строк базы данных повлияла команда.

Метод printResultsTable() обрабатывает вывод результата запроса. Этот метод использует объект ResultSetMetaData, чтобы получить некоторую информацию о возвращенных в результате запроса данных для правильного форматирования результатов.

Необходимо отметить еще два важых приема JDBC-программирования, используемых в примере 17.1. Первым из них является обработка возможных исключений SQLException. Объект SQLException предоставляет стандартное описание исключения с помощью метода getMessage(), но кроме этого может содержать дополнительное сообщение, посланное сервером базы данных. Вы можете получить это сообщение, вызвав метод getSQLState() объекта, представляющего исключение.

Вторым приемом является обработка предупреждений (warning). Класс SQLWarning является производным от класса SQLException, но предупреждения, в отличие от исключений, не генерируются. Во время выполнения SQL-команды все предупреждения, приходящие от сервера, запоминаются в связанном списке объектов SQLWarning. Вы можете получить первый объект SQLWarning этого списка, вызвав метод getWarnings() объекта Connection. Если есть несколько объектов SQLWarning, вы можете получить следующий путем вызова метода getNextWarning() текущего объекта SQLWarning. В примере 17.1 вывод этих предупреждений выполняется при помощи секции finally, таким образом, они по-

явятся и при возникновении исключения, и при нормальном завершении выполнения.

Пример 17.1. ExecuteSQL.java

```
package com.davidflanagan.examples.sql;
import java.sql.*;
import java.io.*;

/**
 * Универсальный интерпретатор команд SQL.
 **/
public class ExecuteSQL {
  public static void main(String[] args) {
    Connection conn = null; // Наше JDBC-соединение с сервером базы данных
    try {
      String driver = null, url = null,
      user = "", password = "";

      // Анализируем все аргументы командной строки
      for(int n = 0; n < args.length; n++) {
        if (args[n].equals("-d")) driver = args[++n];
        else if (args[n].equals("-u")) user = args[++n];
        else if (args[n].equals("-p")) password = args[++n];
        else if (url == null) url = args[n];
        else throw new IllegalArgumentException("Неизвестный аргумент.");
      }

      // Обязательным аргументом является только URL базы данных.
      if (url == null)
        throw new IllegalArgumentException("Не указана база данных");

      // Если пользователь указал имя класса драйвера БД,
      // динамически загружаем этот класс. Это дает драйверу
      // возможность зарегистрировать себя в менеджере
      // драйверов DriverManager.
      if (driver != null) Class.forName(driver);

      // Теперь устанавливаем соединение с указанной базой
      // данных, используя заданные пользователем имя
      // и пароль, если они есть. Менеджер драйверов будет
      // пытаться использовать все известные ему драйверы
      // для анализа URL и соединения с сервером базы данных
      conn = DriverManager.getConnection(url, user, password);

      // Создаем объект Statement, который будем использовать
      // для общения с базой данных
      Statement s = conn.createStatement();

      // Получаем поток для чтения с консоли
      BufferedReader in =
        new BufferedReader(new InputStreamReader(System.in));

      // Бесконечный цикл, считывающий и выполняющий
      // запросы пользователя
```

```
while(true) {
  System.out.print("sql> ");  // приглашение для пользователя
  System.out.flush();         // делаем приглашение видимым
  String sql = in.readLine(); // получаем от пользователя строку ввода
  // Выходим, если пользователь ввел "quit"
  if ((sql == null) || sql.equals("quit")) break;

  // Игнорируем пустые строки
  if (sql.length() == 0) continue;

  // Выполняем пользовательскую SQL-строку и выводим результаты.
  try {
    // Мы не знаем, запрос это или некий тип команды обновления,
    // поэтому вместо executeQuery() или executeUpdate() используем
   // метод execute(). Если возвращенное значение true, это был запрос,
    // в противном случае - команда обновления.
    boolean status = s.execute(sql);

    // Некоторые сложные SQL-запросы могут вернуть
    // несколько итоговых наборов, поэтому выполняем
    // цикл до тех пор, пока не останется результатов
    do {
      if (status) { // Это был запрос, вернувший ResultSet
        // Получаем результаты
        ResultSet rs = s.getResultSet();
        // Выводим их
        printResultsTable(rs, System.out);
      }
      else {
        // Если выполненная SQL-команда была командой обновления,
        // а не запросом, она не вернула ResultSet. Вместо этого мы
        // просто печатаем число строк, на которые повлияла команда.
        int numUpdates = s.getUpdateCount();
       System.out.println("Ok. " + numUpdates + " строк обработано.");
      }

      // Посмотрим, не осталось ли еще результатов,
      // и, если есть, продолжаем их вывод в цикле.
      status = s.getMoreResults();
    } while(status || s.getUpdateCount() != -1);
  }
  // Если генерируется исключение SQLException, выводим сообщение
  // об ошибке. Заметьте, что SQLExceptions может содержать
  // общее сообщение и сообщение, специфичное для БД,
  // возвращаемое методом getSQLState().
  catch (SQLException e) {
    System.err.println("Исключение SQL: " + e.getMessage()+ ":" +
              e.getSQLState());
  }
  // На каждом проходе цикла проверяем наличие предупреждений.
  // Обратите внимание, что здесь может быть целая цепь предупреждений.
  finally { // выводим все поступившие предупреждения
```

```
        SQLWarning w;
        for(w=conn.getWarnings(); w != null; w=w.getNextWarning())
          System.err.println("ПРЕДУПРЕЖДЕНИЕ: " + w.getMessage() +
                     ":" + w.getSQLState());
      }
    }
  }
  // Обрабатываем исключения, возникшие во время анализа
  // аргументов, установки соединения с базой данных и т. д.
  // Для SQLExceptions распечатываем все подробности.
  catch (Exception e) {
    System.err.println(e);
    if (e instanceof SQLException)
      System.err.println("Состояние SQL: " +
                 ((SQLException)e).getSQLState());
    System.err.println("Формат: java ExecuteSQL [-d <driver>] " +
               "[-u <user>] [-p <password>] <database URL>");
  }

  // Гарантируем, что при выходе всегда закрывается соединение с базой
  // данных независимо от того, ввел ли пользователь 'quit' или было
  // сгенерированно исключение. Закрытие этого соединения также неявно
  // закрывает все связанные с ним открытые операторы и итоговые наборы.
      finally {
    try { conn.close(); } catch (Exception e) {}
  }
}

/**
 * Этот метод выполняет попытку вывести содержимое объекта
 * ResultSet в текстовую таблицу. Он рассчитывает на класс
 * ResultSetMetaData, но большая часть кода просто
 * выполняет манипуляции со строками.
 **/
static void printResultsTable(ResultSet rs, OutputStream output)
  throws SQLException
{
  // Настройка выходного потока
  PrintWriter out = new PrintWriter(output);

  // Получаем часть "метаданных" (имена полей и др.)
  // по полученным результатам
  ResultSetMetaData metadata = rs.getMetaData();

  // Переменные для хранения важных данных об отображаемой таблице
  int numcols = metadata.getColumnCount(); // сколько столбцов
  String[] labels = new String[numcols];   // метки столбцов
  int[] colwidths = new int[numcols];      // ширина каждого
  int[] colpos = new int[numcols];         // начальная позиция каждого
  int linewidth;                           // общая ширина таблицы

  // Вычисляем ширину столбцов, начало каждого из них,
  // ширину каждой строки таблицы и т. д.
```

```
linewidth = 1; // до первого символа '|'.
for(int i = 0; i < numcols; i++) {          // для каждого столбца
  colpos[i] = linewidth;                    // сохраняем его позицию
  labels[i] = metadata.getColumnLabel(i+1); // получаем его метку
  // Получаем ширину столбца. Если БД не сообщает ее,
  // принимаем ее равной 30 символам. Затем проверяем
  // длину метки и используем ее, если она превышает ширину столбца
  int size = metadata.getColumnDisplaySize(i+1);
  if (size == -1) size = 30; // некоторые драйверы возвращают -1
  if (size > 500) size = 30; // запрещаем бессмысленные размеры
  int labelsize = labels[i].length();
  if (labelsize > size) size = labelsize;
  colwidths[i] = size + 1;          // запоминаем размер столбца
  linewidth += colwidths[i] + 2;    // увеличиваем общий размер
}

// Создаем горизонтальную разделительную линию, используемую
// в таблице. Также создаем пустую строку, которая будет
// начальным значением для каждой строки таблицы.
StringBuffer divider = new StringBuffer(linewidth);
StringBuffer blankline = new StringBuffer(linewidth);
for(int i = 0; i < linewidth; i++) {
  divider.insert(i, '-');
  blankline.insert(i, " ");
}
// Ставим специальные метки на разделительной линии в позициях столбцов
for(int i=0; i<numcols; i++) divider.setCharAt(colpos[i]-1,'+');
divider.setCharAt(linewidth-1, '+');

// Начинаем вывод таблицы с разделительной линии
out.println(divider);

// Следующая строка таблицы содержит метки столбцов.
// Начинаем с пустой строки и помещаем в нее имена
// столбцов и символы разделения столбцов "|".
// Метод overwrite() определен ниже.
StringBuffer line = new StringBuffer(blankline.toString());
line.setCharAt(0, '|');
for(int i = 0; i < numcols; i++) {
  int pos = colpos[i] + 1 + (colwidths[i]-labels[i].length())/2;
  overwrite(line, pos, labels[i]);
  overwrite(line, colpos[i] + colwidths[i], " |");
}

// Затем выводим строку с метками столбцов и другими разделителями
out.println(line);
out.println(divider);

// Теперь выводим табличные данные. Выполняем цикл
// по ResultSet, используя метод next() для получения
// строк по одной за раз. Получаем значение каждого
// столбца вызовом getObject() и выводим его почти
// таким же способом, как метки столбцов.
```

```
      while(rs.next()) {
        line = new StringBuffer(blankline.toString());
        line.setCharAt(0, '|');
        for(int i = 0; i < numcols; i++) {
          Object value = rs.getObject(i+1);
          if (value != null)
            overwrite(line, colpos[i] + 1, value.toString().trim());
          overwrite(line, colpos[i] + colwidths[i], " |");
        }
        out.println(line);
      }

      // И в конце завершаем таблицу последней разделительной линией.
      out.println(divider);
      out.flush();
    }

    /** Этот вспомогательный метод используется при печати таблицы результатов */
    static void overwrite(StringBuffer b, int pos, String s) {
      int slen = s.length();                // Длина строки
      int blen = b.length();                // Длина буфера
      if (pos+slen > blen) slen = blen-pos; // Входит ли она?
      for(int i = 0; i < slen; i++)         // Копируем строку в буфер
        b.setCharAt(pos+i, s.charAt(i));
    }
  }
```

Использование метаданных базы данных

Иногда кроме запроса и обновления данных в базе данных вам будет нужно получить информацию о самой базе данных и ее содержимом. Эта информация называется *метаданными* (*metadata*). Интерфейс `DatabaseMetaData` позволяет вам получать этот тип информации. Вы можете получить объект, реализующий этот интерфейс, путем вызова метода `getMetaData()` объекта `Connection`, как показано в примере 17.2.

После того как программа `GetDBInfo` установит соединение с базой данных и получит объект `DatabaseMetaData`, она выводит некоторую общую информацию о сервере базы данных и JDBC-драйвере. Затем, если пользователь просто укажет в командной строке имя базы данных, программа отобразит список всех таблиц, находящихся в этой базе данных. А если пользователь укажет имя базы данных вместе с именем таблицы, программа выведет список имен и типов данных полей этой таблицы.

Интересная особенность программы `GetDBInfo` состоит в том, как она получает параметры, необходимые для соединения с базой данных. Пример основывается на допущении, что с каждого отдельного рабочего места обычно подключаются к одному и тому же серверу баз данных, используя один и тот же драйвер, а возможно, что и одни и те же имя

пользователя и пароль. Поэтому, вместо того чтобы при каждом запуске программы просить пользователя вводить в командной строке все эти многочисленные сведения, программа считывает для них значения по умолчанию из файла *DB.props*, находящегося в одном каталоге с файлом *GetDBInfo.class*. Для того чтобы выполнить пример 17.2, вам необходимо создать соответствующий вашей системе файл *DB.props*. В моей системе этот файл выглядит так:

```
# Имя класса JDBC-драйвера
driver=org.gjt.mm.mysql.Driver
# URL, определяющий сервер базы данных.
# Он не должен содержать имя используемой базы данных
server=jdbc:mysql://db.domain.com/
# Имя учетной записи в базе данных
user=david
# Пароль для указанной учетной записи. Здесь не указан
# Раскомментируйте следующую строку для задания пароля
#password=
```

Ясно, что строки, начинающиеся с символа #, являются комментариями. Формат *имя=значение* является стандартным файловым форматом для объекта `java.util.Properties`, который используется для считывания содержимого этого файла.

После того как программа считает значения по умолчанию из файла *DB.props*, она анализирует аргументы командной строки, которые могут заместить значения свойств `driver`, `server`, `user` и `password`, заданные в этом файле. Имя используемой базы данных должно быть задано в командной строке, оно просто добавляется в конец URL сервера. В командной строке может быть задано необязательное имя таблицы. Например, вы можете запустить программу так:

```
% java com.davidflanagan.examples.sql.GetDBInfo api class
DBMS: MySQL 3.22.32
JDBC-драйвер: Mark Matthews' MySQL Driver 2.0a
База данных: jdbc:mysql://localhost/api
Пользователь: david
Поля таблицы class:
    id : int
    packageId : int
    name : varchar
```

Пример 17.2. GetDBInfo.java

```
package com.davidflanagan.examples.sql;
import java.sql.*;
import java.util.Properties;

/**
 * Этот класс использует класс DatabaseMetaData для получения информации о базе
 * данных, JDBC-драйвере и таблицах базы данных или о полях заданной таблицы.
 **/
```

```
public class GetDBInfo {
  public static void main(String[] args) {
    Connection c = null; // JDBC-соединение с сервером базы данных
    try {
      // Ищем файл свойств DB.props в том же каталоге, в котором расположена
      // программа. Он содержит значения по умолчанию различных параметров,
      // необходимых для соединения с базой данных
      Properties p = new Properties();
      try { p.load(GetDBInfo.class.getResourceAsStream("DB.props")); }
      catch (Exception e) {}

      // Получаем из файла свойств значения по умолчанию
      String driver = p.getProperty("driver");        // Имя класса драйвера
      String server = p.getProperty("server", "");    // URL для сервера
      String user = p.getProperty("user", "");        // Имя пользователя
      String password = p.getProperty("password", ""); // пароль

      // У этих переменных нет значения по умолчанию
      String database = null; // Имя базы данных (добавляется к URL сервера)
      String table = null;    // Необязательное имя таблицы в БД

      // Анализируем параметры командной строки, замещающие
      // считанные выше значения по умолчанию
      for(int i = 0; i < args.length; i++) {
        if (args[i].equals("-d")) driver = args[++i];      //-d <driver>
        else if (args[i].equals("-s")) server = args[++i]; //-s <server>
        else if (args[i].equals("-u")) user = args[++i];   //-u <user>
        else if (args[i].equals("-p")) password = args[++i];
        else if (database == null) database = args[i];     // <dbname>
        else if (table == null) table = args[i];           // <table>
        else throw new IllegalArgumentException(
                              "Неизвестный параметр: "+args[i]);
      }

      // Проверяем, что, по крайней мере, заданы сервер и база данных.
      // Если нет, то мы не знаем, с чем нужно соединяться,
      // и не можем продолжить.
      if ((server.length() == 0) && (database.length() == 0))
        throw new IllegalArgumentException("Не указана база данных.");

      // Загружаем драйвер БД, если он был указан
      if (driver != null) Class.forName(driver);

      // Пытаемся открыть соединение с указанной базой данных
      // на указанном сервере, используя указанные имя и пароль
      c = DriverManager.getConnection(server+database, user, password);

      // Получаем объект DatabaseMetaData для этого соединения. Это объект,
      // из которого мы получим всю необходимую нам информацию.
      DatabaseMetaData md = c.getMetaData();

      // Выводим информацию о сервере, драйвере и др.
      System.out.println("DBMS: " + md.getDatabaseProductName() +
                " " + md.getDatabaseProductVersion());
```

```
        System.out.println("JDBC-драйвер: " + md.getDriverName() +
                  " " + md.getDriverVersion());
        System.out.println("База данных: " + md.getURL());
        System.out.println("Пользователь: " + md.getUserName());

        // Если пользователь не указал таблицу, выводим список
        // всех таблиц, определенных в заданной базе данных.
        // Заметим, что таблицы возвращаются в объекте
        // ResultSet, как и результаты запроса.
        if (table == null) {
          System.out.println("Таблицы:");
          ResultSet r = md.getTables("", "", "%", null);
          while(r.next()) System.out.println("\t" + r.getString(3));
        }

        // В противном случае выводим список всех полей
        // указанной таблицы. И снова информация о полях
        // возвращается в объекте ResultSet.
        else {
          System.out.println("Поля таблицы " + table + ": ");
          ResultSet r = md.getColumns("", "", table, "%");
          while(r.next())
            System.out.println("\t" + r.getString(4) + " : " +
                    r.getString(6));
        }
      }
      // При возникновении ошибки выводим сообщение.
      catch (Exception e) {
        System.err.println(e);
        if (e instanceof SQLException)
          System.err.println(((SQLException)e).getSQLState());
        System.err.println("Формат: java GetDBInfo [-d <driver] " +
                  "[-s <dbserver>]\n" +
                  "\t[-u <username>] [-p <password>] <dbname>");
      }
      // Никогда не забываем в конце закрыть объект Connection!
      finally {
        try { c.close(); } catch (Exception e) {}
      }
    }
  }
```

Создание базы данных

В примере 17.3 показана программа `MakeAPIDB`, которая принимает список имен классов и использует API отражения Java для создания базы данных из этих классов, пакетов, которым они принадлежат, и всех методов и полей, определенных в этих классах. В примере 17.4 приведена программа, которая использует базу данных, созданную в этом примере.

В MakeAPIDB используется SQL-оператор CREATE TABLE для добавления в базу данных трех таблиц с именами package, class и member. Затем программа при помощи операторов INSERT INTO вставляет данные в эти таблицы. В процессе итерации по списку имен классов программа повторно использует один и тот же оператор INSERT INTO. В подобных ситуациях вы зачастую можете увеличить эффективность таких вставок путем применения объектов PreparedStatement для выполнения этих операторов.

Подготовленный оператор (prepared statement), по существу, является заготовкой для операторов, которые вам необходимо выполнить. Когда вы отсылаете базе данных SQL-оператор, она интерпретирует этот SQL-текст и создает шаблон для выполнения оператора. Если вы многократно посылаете один и тот же SQL-оператор, отличающийся лишь входными параметрами, база данных будет интерпретировать его каждый раз. Используя платформы баз данных с поддержкой подготовленных операторов, вы можете устранить эту неэффективность, отослав базе данных подготовленный оператор перед выполнением действительных вызовов. База данных интерпретирует подготовленный оператор и создает его шаблон только один раз. Затем, при многократном выполнении подготовленного оператора с различными входными параметрами, база данных использует этот заранее созданный ею шаблон. Для поддержки подготовленных операторов в JDBC предусмотрен класс PreparedStatement, но нет никакой гарантии, что используемая база данных поддерживает этот тип операторов.

Как показано в примере 17.3, для создания объекта PreparedStatement используется метод prepareStatement() объекта Connection. MakeAPIDB передает SQL-оператор в метод prepareStatement(), заменяя символом подстановки ? переменные параметры в этом операторе. Позднее, перед выполнением подготовленного оператора, программа связывает с этими параметрами действительные значения при помощи методов set*X*() (то есть setInt() и setString()) объекта PreparedStatement. Каждый метод set*X*() принимает два аргумента: индекс параметра (начиная с 1) и его значение. Затем программа вызывает метод executeUpdate() объекта PreparedStatement для выполнения оператора. (Так же как объект Statement, PreparedStatement предоставляет методы execute() и executeQuery()).

Программа MakeAPIDB ожидает, что ее первым параметром будет имя файла, содержащего список классов, которые требуется занести в базу данных. Эти классы должны быть перечислены по одному в строке, и каждая строка должна содержать полное имя класса (то есть должны быть указаны как имя пакета, так и имя класса). Такой файл может содержать, например, такие строки:

```
java.applet.Applet
java.applet.AppletContext
java.applet.AppletStub
    ...
java.util.zip.ZipOutputStream
```

Программа считывает параметры, связанные с базой данных, из файла свойств с именем *APIDB.props*, находящегося в текущем каталоге, или из альтернативного файла, указанного в качестве второго параметра командной строки. Данный файл свойств похож, но не совсем идентичен тому, который использовался совместно с примером 17.2. Он должен содержать свойства с именами `driver`, `database`, `user` и `password`. В моей системе файл *APIDB.props* выглядит так:

```
# Полное имя класса загружаемого JDBC-драйвера:
# это драйвер для MySQL
driver=org.gjt.mm.mysql.Driver
# URL сервера mysql (localhost) и базы данных (apidb)
database=jdbc:mysql:///apidb
# Имя учетной записи базы данных
user=david
# Пароль для учетной записи базы данных.
# Раскомментируйте следующую строку для задания пароля
#password=
```

Обратите внимание, что перед запуском программы вам необходимо создать для нее базу данных на вашем сервере. Для того чтобы сделать это, следуйте инструкциям, предоставленным поставщиком вашей базы данных. Вы также можете использовать существующую базу данных, если она не содержит таблиц `package`, `class` и `member`.[1]

Пример 17.3. MakeAPIDB.java

```
package com.davidflanagan.examples.sql;
import java.sql.*;
import java.lang.reflect.*;
import java.io.*;
import java.util.*;

/**
 * Этот класс является самостоятельной программой, которая
 * считывает список классов и создает базу данных пакетов,
 * классов, а также их полей и методов.
 **/
public class MakeAPIDB {
 public static void main(String args[]) {
 Connection c = null; // Соединение с базой данных
 try {
```

[1] В программе выполняется большое количество обновлений базы данных. В моих тестах этой программы сервер MySQL был значительно быстрее, чем сервер PostgreSQL. Кроме того, PostgreSQL не распознает стандартный тип `BIT` (на самом деле тип `BIT` не поддерживается JDBC-драйвером PostgreSQL, сама база поддерживает тип `BOOLEAN` для хранения подобных значений. – *Примеч. науч. ред.*), используемого в таблице `member`. Если с этим примером вы планируете использовать сервер PostgreSQL, вам нужно будет в этой таблице изменить тип поля `isField`.

```
// Формируем указатель классов, считывая их из файла,
// заданного аргументом args[0]
ArrayList classnames = new ArrayList();
BufferedReader in = new BufferedReader(new FileReader(args[0]));
String name;
while((name = in.readLine()) != null) classnames.add(name);

// Сейчас находим значения, необходимые для установления
// соединения с базой данных. Программа пытается считать
// файл свойств с именем APIDB.props или заданный
// необязательным агрументом args[1]. Этот файл свойств
// (если есть) может содержать свойства "driver",
// "database", "user" и "password", задающие параметры,
// необходимые для соединения с базой данных. Если файл
// свойств не существует или не содержит нужных параметров,
// то будут использоваться значения по умолчанию.
Properties p = new Properties();  // Пустые свойства
// Пытаемся загрузить свойства
try {
  p.load(new FileInputStream(args[1]));
}
catch (Exception e1) {
try { p.load(new FileInputStream("APIDB.props")); }
catch (Exception e2) {}
}

// Считываем значения из файла свойств
String driver = p.getProperty("driver");
String database = p.getProperty("database");
String user = p.getProperty("user", "");
String password = p.getProperty("password", "");

// Свойства driver и database являются обязательными
if (driver == null)
throw new IllegalArgumentException("Драйвер не задан!");
if (database == null)
throw new IllegalArgumentException("База данных не задана!");

// Загружаем драйвер. Он регистрирует себя в DriverManager.
Class.forName(driver);

// И устанавливаем соединение с заданной базой данных
c = DriverManager.getConnection(database, user, password);

// Создаем для наших данных три новые таблицы.
// Таблица package содержит идентификаторы и имена пакетов.
// Таблица class содержит идентификаторы классов, пакетов
// и имена классов. Таблица member содержит идентификаторы
// классов, имена членов и бит, указывающий, чем является
// член класса: полем или методом.
Statement s = c.createStatement();
s.executeUpdate("CREATE TABLE package " +
  "(id INT, name VARCHAR(80))");
s.executeUpdate("CREATE TABLE class " +
```

```
    "(id INT, packageId INT, name VARCHAR(48))");
  s.executeUpdate("CREATE TABLE member " +
    "(classId INT, name VARCHAR(48), isField BIT)");

  // Подготавливаем несколько операторов, которые будут
  // использоваться для вставки записей в эти три таблицы
  insertpackage =
  c.prepareStatement("INSERT INTO package VALUES(?,?)");
  insertclass =
  c.prepareStatement("INSERT INTO class VALUES(?,?,?)");
  insertmember =
  c.prepareStatement("INSERT INTO member VALUES(?,?,?)");

  // Выполняем цикл по всем классам и используем механизм
  // отражения для их записи в таблицы
  int numclasses = classnames.size();
  for(int i = 0; i < numclasses; i++) {
  try {
   storeClass((String)classnames.get(i));
  }
  catch(ClassNotFoundException e) {
   System.out.println("ПРЕДУПРЕЖДЕНИЕ: класс не найден: " +
     classnames.get(i) + "; ПРОПУЩЕН");
  }
  }
}
catch (Exception e) {
 System.err.println(e);
 if (e instanceof SQLException)
 System.err.println("SQLState: " +
    ((SQLException)e).getSQLState());
 System.err.println("Формат: java MakeAPIDB " +
   "<classlistfile> <propfile>");
}
// После завершения закрываем соединение с базой данных
finally { try { c.close(); } catch (Exception e) {} }
}

/**
* Эта хеш-таблица хранит соответствия между именами
* и идентификаторами пакетов. Это все, что нам нужно
* для временного хранения. Все остальное запоминается в БД
* и не подлежит поиску из этой программы.
**/
static Map package_to_id = new HashMap();

// Счетчики для полей идентификаторов пакетов и классов
static int packageId = 0, classId = 0;

// Несколько подготовленных SQL-операторов, используемых
// для вставки в таблицы новых значений.
// Были проинициализированы выше в методе main().
static PreparedStatement insertpackage, insertclass, insertmember;
```

```
/**
* Получая полное имя класса, этот метод записывает имя пакета в таблицу
* package (если его еще там нет), имя класса – в таблицу class,
* а затем при помощи интерфейса отражения Java находит все методы
* и поля класса и записывает их в таблицу member.
**/
public static void storeClass(String name)
throws SQLException, ClassNotFoundException
{
String packagename, classname;

// Динамически загружаем класс.
Class c = Class.forName(name);

// Выводим сообщения для того, чтобы пользователь знал,
// что программа работает
System.out.println("Сохраняются данные для: " + name);

// Определяем имена пакета и класса
int pos = name.lastIndexOf('.');
if (pos == -1) {
 packagename = "";
 classname = name;
}
else {
 packagename = name.substring(0,pos);
 classname = name.substring(pos+1);
}

// Выясняем, есть ли у данного пакета идентификатор.
// Если есть, то этот пакет уже находится в базе данных.
// В противном случае присваиваем ему идентификатор
// и записываем его имя в базу данных.
Integer pid;
// Проверяем хеш-таблицу
pid = (Integer)package_to_id.get(packagename);
if (pid == null) {
 pid = new Integer(++packageId);        // Присваиваем id
 package_to_id.put(packagename, pid);   // Запоминаем его
 insertpackage.setInt(1, packageId);    // Задаем оператору аргументы
 insertpackage.setString(2, packagename);
 insertpackage.executeUpdate();         // Вставляем в таблицу package
}

// Теперь записываем имя класса в таблицу class базы данных. В эту запись
// входит идентификатор пакета id, поэтому класс связан с содержащим его
// пакетом. Для записи класса мы задаем аргументы подготовленного оператора
// PreparedStatement, а затем выполняем его.
 insertclass.setInt(1, ++classId);      // Устанавливаем id класса
insertclass.setInt(2, pid.intValue()); // id пакета
insertclass.setString(3, classname);   // Имя класса
insertclass.executeUpdate();           // Вставляем запись класса

// Теперь получаем список всех незакрытых методов класса
```

```
        // и вставляем их в таблицу member базы данных.
        // Каждая запись содержит идентификатор содержащего ее класса,
        // а также значение, указывающее, является она методом или полем
        Method[] methods = c.getDeclaredMethods(); // Получаем список методов
        for(int i = 0; i < methods.length; i++) {  // По всем незакрытым
         if (Modifier.isPrivate(methods[i].getModifiers())) continue;
         insertmember.setInt(1, classId);          // Задаем id класса
         insertmember.setString(2, methods[i].getName()); // Имя метода
         insertmember.setBoolean(3, false);        // Это не поле
         insertmember.executeUpdate();             // Вставляем в БД
        }

        // Тоже самое делаем и для незакрытых полей класса
        Field[] fields = c.getDeclaredFields();    // Список полей
        for(int i = 0; i < fields.length; i++) {   // По всем незакрытым
         if (Modifier.isPrivate(fields[i].getModifiers())) continue;
         insertmember.setInt(1, classId);          // Задаем id класса
         insertmember.setString(2, fields[i].getName()); // Имя поля
         insertmember.setBoolean(3, true);         // Это поле
         insertmember.executeUpdate();             // Вставляем запись
        }
        }
    }
```

Использование API баз данных

В примере 17.4 показана программа `LookupAPI`, которая использует базу данных, созданную программой `MakeAPIDB` из примера 17.3. `LookupAPI` работает следующим образом:

- При вызове с именем члена класса она выводит полные имена (включая пакет и класс) всех полей и методов с таким именем.
- При запуске с именем класса она выводит список полных имен всех классов из всех пакетов с таким именем.
- При указании части имени пакета она выводит список имен всех пакетов, которые содержат эту строку.
- При вызове с ключом `-l` и именем класса она выводит все члены всех классов, имеющих это имя.
- При запуске с ключом `-l` и частью имени пакета она выводит список всех классов и интерфейсов из всех пакетов, соответствующих этой строке.

`LookupAPI` использует тот же файл свойств *APIDB.props*, что и программа `MakeAPIDB`. Или, в качестве альтернативного, считывает файл свойств, указанный в командной строке после ключа `-p`. Используя параметры соединения из файла свойств, программа соединяется с базой данных и выполняет SQL-запросы, необходимые для получения требуемой информации. Обратите внимание, что она вызывает метод `setReadOnly()` объекта `Connection`. Это подсказывает, что программа выполняет толь-

ко запросы и никаким образом не модифицирует базу данных. В некоторых системах баз данных это может улучшить производительность. Кроме метода `setReadOnly()`, в этом примере не представлено никаких новых возможностей JDBC. Он просто служит в качестве реального приложения баз данных и демонстрирует некоторые мощные запросы базы данных, которые могут быть выражены с помощью SQL.

Пример 17.4. LookupAPI.java

```
package com.davidflanagan.examples.sql;
import java.sql.*;
import java.io.FileInputStream;
import java.util.Properties;

/**
 * Эта программа использует базу данных, созданную программой MakeAPIDB.
 * Она открывает соединение с базой данных, используя тот же
 * файл свойств, что и MakeAPIDB. Затем она запрашивает базу
 * данных несколькими интересными способами для получения
 * необходимой информации по Java API. Она может применяться
 * для поиска полных имен членов, классов и пакетов или
 * для вывода списка членов класса или пакета.
 **/
public class LookupAPI {
 public static void main(String[] args) {
 Connection c = null;  // JDBC-соединение с базой данных
 try {
  // Несколько значений по умолчанию
  String target = null;  // Имя для поиска
  boolean list = false;  // Вывести члены или найти имя?
  String propfile = "APIDB.props"; // файл параметров БД

  // Анализ аргументов командной строки
  for(int i = 0; i < args.length; i++) {
  if (args[i].equals("-l")) list = true;
  else if (args[i].equals("-p")) propfile = args[++i];
  else if (target != null)
   throw new IllegalArgumentException(
     "Неизвестный аргумент: " + args[i]);
  else target = args[i];
  }
  if (target == null)
  throw new IllegalArgumentException("Целевой объект не задан");

  // Теперь определяем значения, необходимые для установки
  // соединения с базой данных. Программа пробует считать
  // файл свойств с именем APIDB.props или указанный
  // с помощью необязательного ключа -p. Этот файл свойств
  // может содержать свойства “driver”, “user” и “password”,
  // задающие значения, необходимые для соединения с БД.
  // Если этот файл свойств не существует или не содержит
  // указанных свойств, будут использоваться значения по умолчанию.
```

```
 Properties p = new Properties();  // Пустые свойства
 // Пытаемся загрузить свойства
 try { p.load(new FileInputStream(propfile)); }
 catch (Exception e) {}

 // Считываем значения из файла свойств
 String driver = p.getProperty("driver");
 String database = p.getProperty("database");
 String user = p.getProperty("user", "");
 String password = p.getProperty("password", "");

 // Свойства driver и database являются обязательными
 if (driver == null)
 throw new IllegalArgumentException("Драйвер не указан!");
 if (database == null)
 throw new IllegalArgumentException("База данных не указана!");

 // Загрузка драйвера базы данных
 Class.forName(driver);

 // Устанавливаем соединение с указанной базой данных
 c = DriverManager.getConnection(database, user, password);

 // Сообщаем ей, что мы не будем делать обновлений.
 // Это может улучшить производительность.
 c.setReadOnly(true);

 // Если был задан ключ -l, то выводим список членов
 // указанного пакета или класса. В противном случае ищем
 // все совпадения с указанным членом, классом или пакетом.
 if (list) list(c, target);
 else lookup(c, target);
}
// При возникновении ошибки выводим исключение и сообщение
// об использовании программы. Если генерируется исключение
// SQLException, выводим входящее в него сообщение о состоянии.
catch (Exception e) {
 System.out.println(e);
 if (e instanceof SQLException)
 System.out.println(((SQLException) e).getSQLState());
 System.out.println("Формат: java LookupAPI [-l] [-p <propfile>] " +
   "target");
}
// По завершении работы с БД-соединением всегда закрываем его.
finally {
 try { c.close(); } catch (Exception e) {}
}
}

/**
* Этот метод ищет в базе данных все совпадения с указанной целевой строкой.
* Сначала он выводит полное имя всех членов с таким именем. Затем печатает
* полные имена всех классов с таким именем. Потом выводит имена всех
* пакетов, которые содержат указанное имя.
```

```
**/
public static void lookup(Connection c, String target) throws SQLException
{
// Создаем объект Statement, который мы будем использовать
// для запросов к базе данных
Statement s = c.createStatement();

// Идем искать все члены классов с указанным именем
s.executeQuery("SELECT DISTINCT " +
  "package.name, class.name, member.name, member.isField"+
  " FROM package, class, member" +
  " WHERE member.name='" + target + "'" +
  " AND member.classId=class.id " +
  " AND class.packageId=package.id");

// Выполняем цикл по результатам и распечатываем их (если они есть).
ResultSet r = s.getResultSet();
while(r.next()) {
 String pkg = r.getString(1);       // имя пакета
 String cls = r.getString(2);       // имя класса
 String member = r.getString(3);    // имя члена
 boolean isField = r.getBoolean(4); // член является полем?
 // Выводим это совпадение
 System.out.println(pkg + "." + cls + "." + member +
   (isField?"":"()"));
}

// Сейчас ищем класс с заданным именем
s.executeQuery("SELECT package.name, class.name " +
  "FROM package, class " +
  "WHERE class.name='" + target + "' " +
  " AND class.packageId=package.id");
// Выполняем цикл по результатам и распечатываем их
r = s.getResultSet();
while(r.next()) System.out.println(r.getString(1) + "." +
     r.getString(2));

// Наконец, ищем пакет, в имени которого содержится
// заданный фрагмент. Обратите внимание на использование
// ключевого слова LIKE и символов подстановки %
s.executeQuery("SELECT name FROM package " +
  "WHERE name='" + target + "'" +
  " OR name LIKE '%." + target + ".%' " +
  " OR name LIKE '" + target + ".%' " +
  " OR name LIKE '%." + target + "'");
// Выполняем цикл по результатам и распечатываем их.
r = s.getResultSet();
while(r.next()) System.out.println(r.getString(1));

// И наконец, закрываем объект Statement
s.close();
}

/**
```

```
* Этот метод ищет классы с заданным именем или пакеты,
* содержащие заданное имя. Для каждого найденного класса
* он отображает все его методы и поля. Для каждого
* найденного пакета он отображает все классы пакета.
**/
public static void list(Connection conn, String target) throws SQLException
{
// Создаем два объекта Statement для запроса базы данных
Statement s = conn.createStatement();
Statement t = conn.createStatement();

// Ищем класс с заданным именем
s.executeQuery("SELECT package.name, class.name " +
  "FROM package, class " +
  "WHERE class.name='" + target + "' " +
  " AND class.packageId=package.id");
// Цикл по всем совпадениям
ResultSet r = s.getResultSet();
while(r.next()) {
 String p = r.getString(1); // имя пакета
 String c = r.getString(2); // имя класса
 // Распечатываем имя найденного класса
 System.out.println("class " + p + "." + c + " {");

 // Теперь запрашиваем все члены этого класса
 t.executeQuery("SELECT DISTINCT member.name, member.isField " +
   "FROM package, class, member " +
   "WHERE package.name = '" + p + "' " +
   " AND class.name = '" + c + "' " +
   " AND member.classId=class.id " +
   " AND class.packageId=package.id " +
   "ORDER BY member.isField, member.name");

 // Выполняем цикл по отсортированному списку всех членов и распечатываем их
 ResultSet r2 = t.getResultSet();
 while(r2.next()) {
 String m = r2.getString(1);
 int isField = r2.getInt(2);
 System.out.println(" " + m + ((isField == 1)?"":"()"));
 }
 // Завершаем список классов
 System.out.println("}");
}

// Теперь выполняем поиск пакетов, удовлетворяющих заданному имени
s.executeQuery("SELECT name FROM package " +
  "WHERE name='" + target + "'" +
  " OR name LIKE '%." + target + ".%' " +
  " OR name LIKE '" + target + ".%' " +
  " OR name LIKE '%." + target + "'");
// Цикл по всем найденным пакетам
r = s.getResultSet();
while(r.next()) {
```

```
        // Выводим имя пакета
        String p = r.getString(1);
        System.out.println("Package " + p + ": ");

        // Получаем список всех классов и интерфейсов пакета
        t.executeQuery("SELECT class.name FROM package, class " +
          "WHERE package.name='" + p + "' " +
          " AND class.packageId=package.id " +
          "ORDER BY class.name");
        // Выполняем цикл по списку и распечатываем их.
        ResultSet r2 = t.getResultSet();
        while(r2.next()) System.out.println(" " + r2.getString(1));
      }

      // В конце закрываем оба объекта Statement
      s.close(); t.close();
      }
    }
```

Атомарные транзакции

По умолчанию для вновь созданного объекта `Connection` устанавливается режим автоматической фиксации (auto-commit mode). Это значит, что каждое обновление базы данных считается отдельной транзакцией и автоматически фиксируется в базе данных. Однако иногда вам может потребоваться сгруппировать несколько обновлений в одну атомарную (неделимую, atomic) транзакцию, обладающую тем свойством, что либо все ее обновления заканчиваются успешно, либо не происходят вовсе. При работе с системой баз данных (и JDBC-драйвером), которая поддерживает их, вы можете вывести объект `Connection` из режима автофиксации и явно вызывать метод `commit()` для фиксации пакета транзакций или метод `rollback()` для прерывания пакета транзакций, отменяя те, которые уже были выполнены.

В примере 17.5 показан класс, использующий атомарные транзакции с целью обеспечения согласованности базы данных. Он представляет собой реализацию интерфейса `RemoteBank`, который был разработан в главе 16 «Вызов удаленных методов (RMI)». Как вы, может быть, помните, созданный в той главе класс `RemoteBankServer` не предоставлял для данных, относящихся с банковскому счету, никакого вида постоянного хранения. В примере 17.5 эта проблема решается путем реализации `RemoteBank` с использованием базы данных для хранения всей информации о клиентских счетах.

После создания объекта `Connection` объект `RemoteDBBankServer` вызывает метод `setAutoCommit()` с параметром `false`, что приводит к отключению режима автофиксации. Затем, к примеру, метод `openAccount()` группирует три транзакции в одну атомарную транзакцию: добавление счета в базу данных, создание таблицы для истории счета и добавление в историю начальной записи. Если все три транзакции завершаются ус-

пешно (то есть не генерируют никаких исключений), `openAccount()` вызывает `commit()` для фиксации этих транзакций в базе данных. Однако если хотя бы одна из этих транзакций возбуждает исключение, секция `catch` позаботится о вызове метода `rollback()` для выполнения отката всех успешных транзакций. Все удаленные методы объекта `RemoteDBBankServer` используют этот подход для сохранения согласованности базы данных счетов.

Помимо демонстрации приемов обработки атомарных транзакций `RemoteDBBankServer` дает дополнительные примеры использования SQL-запросов при взаимодействии с базой данных. Для того чтобы выполнить этот пример, вам потребуется создать файл свойств с именем *BankDB.props*, содержащий информацию о подключении к базе данных. Кроме этого, перед первым запуском сервера вам необходимо создать в базе данных таблицу `account`. Вы можете сделать это с помощью программы `ExecuteSQL`, предназначенной для выполнения этого SQL-оператора.

```
CREATE TABLE accounts (name VARCHAR(20), password VARCHAR(20), balance INT);
```

Не забывайте и про сервер RMI. Это значит, что вы должны запустить компилятор *rmic*, как это было описано в главе 16, для генерации классов каркаса и заглушки. И еще перед запуском этой программы вы должны запустить службу *rmiregistry*.[1]

Пример 17.5. RemoteDBBankServer.java

```
package com.davidflanagan.examples.sql;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.sql.*;
import java.io.*;
import java.util.*;
// Импортируем явно для разрешения неоднозначности с java.sql.Date
import java.util.Date;
// Импортируем внутренние классы класса Bank
import com.davidflanagan.examples.rmi.Bank.*;

/**
 * Этот класс представляет собой еще одну реализацию интерфейса RemoteBank.
 * Он использует в качестве источника данных соединение с базой данных,
 * благодаря чему при выключении сервера не происходит потеря клиентских
 * данных. Обратите внимание, что у соединения с базой данных отключается
 * режим автофиксации, а для обеспечения атомарного выполнения обновлений
 * выполняется явный вызов методов commit() и rollback().
```

[1] Заметьте также, что в сервере MySQL до версии 2.23.15 не была реализована поддержка транзакций. Если вы используете одну из ранних его версий, то для того чтобы запустить этот пример, вам понадобится обновить его или перейти на другую базу данных.

```
**/
public class RemoteDBBankServer extends UnicastRemoteObject
 implements RemoteBank
{
 // Соединение с базой данных, хранящей информацию о счетах
 Connection db;

 /** Конструктор. Просто запоминает соединение с БД */
 public RemoteDBBankServer(Connection db) throws RemoteException {
 this.db = db;
 }

 /** Открытие счета */
 public synchronized void openAccount(String name, String password)
 throws RemoteException, BankingException
 {
 // Прежде всего, проверяем, открыт ли уже счет с таким именем
 Statement s = null;
 try {
  s = db.createStatement();
  s.executeQuery("SELECT * FROM accounts WHERE name='" + name + "'");
  ResultSet r = s.getResultSet();
  if (r.next()) throw new BankingException("Имя счета занято.");

  // Если оно не существует, идем дальше и создаем его.
  // А также создаем таблицу для истории проводок по этому
  // счету и вставляем в нее начальную проводку
  s = db.createStatement();
  s.executeUpdate("INSERT INTO accounts VALUES ('" + name + "', '" +
    password + "', 0)");
  s.executeUpdate("CREATE TABLE " + name +
    "_history (msg VARCHAR(80))");
  s.executeUpdate("INSERT INTO " + name + "_history " +
    "VALUES ('Счет открыт " + new Date() + "')");

  // И если до сих пор все выполнялось успешно, фиксируем
  // эти обновления, завершая атомарную транзакцию.
  // Все расположенные ниже методы также используют эту схему
  // фиксации/отката атомарных транзакций
  db.commit();
 }
 catch(SQLException e) {
  // Если произошло исключение, выполняем откат предыдущих обновлений,
  // удаляя их из базы данных. Это также завершает атомарную транзакцию
  try { db.rollback(); } catch (Exception e2) {}
  // Передаем SQLException в виде BankingException
  throw new BankingException("Исключение SQL: " + e.getMessage() +
     ": " + e.getSQLState());
 }
 // Независимо от того, что произошло, закрываем Statement
 finally { try { s.close(); } catch (Exception e) {} }
 }
```

```
/**
* Этот служебный метод проверяет, соответствуют ли имя
* и пароль какому-нибудь существующему счету. Если это так,
* то он возвращает баланс этого счета. А если нет, то он
* генерирует исключение. Заметим, что в этом методе
* не вызываются ни commit(), ни rollback(), поэтому
* его запрос является частью более крупной транзакции.
**/
public int verify(String name, String password)
throws BankingException, SQLException
{
Statement s = null;
try {
 s = db.createStatement();
 s.executeQuery("SELECT balance FROM accounts " +
   "WHERE name='" + name + "' " +
   " AND password = '" + password + "'");
 ResultSet r = s.getResultSet();
 if (!r.next())
 throw new BankingException("Неверное имя счета или пароль");
 return r.getInt(1);
}
finally { try { s.close(); } catch (Exception e) {} }
}

/** Закрываем указанный счет */
public synchronized FunnyMoney closeAccount(String name, String password)
throws RemoteException, BankingException
{
int balance = 0;
Statement s = null;
try {
 balance = verify(name, password);
 s = db.createStatement();
 // Удаляем счет из таблицы счетов
 s.executeUpdate("DELETE FROM accounts " +
   "WHERE name = '" + name + "' " +
   " AND password = '" + password + "'");
 // И очищаем историю транзакций для этого счета
 s.executeUpdate("DROP TABLE " + name + "_history");
 db.commit();
}
catch (SQLException e) {
 try { db.rollback(); } catch (Exception e2) {}
 throw new BankingException("SQLException: " + e.getMessage() +
    ": " + e.getSQLState());
}
finally { try { s.close(); } catch (Exception e) {} }

// Возвращаем весь баланс, оставшийся на счете
return new FunnyMoney(balance);
}
```

```
 /** Вносим указанную денежную сумму на заданный счет */
public synchronized void deposit(String name, String password,
    FunnyMoney money)
throws RemoteException, BankingException
{
int balance = 0;
Statement s = null;
try {
 balance = verify(name, password);
 s = db.createStatement();
 // Обновляем баланс
 s.executeUpdate("UPDATE accounts " +
   "SET balance = " + balance + money.amount + " " +
   "WHERE name='" + name + "' " +
   " AND password = '" + password + "'");
 // Добавляем строку к истории проводок
 s.executeUpdate("INSERT INTO " + name + "_history " +
   "VALUES ('Внесено " + money.amount +
   " в " + new Date() + "')");
 db.commit();
}
catch (SQLException e) {
 try { db.rollback(); } catch (Exception e2) {}
 throw new BankingException("Исключение SQL: " + e.getMessage() +
    ": " + e.getSQLState());
}
finally { try { s.close(); } catch (Exception e) {} }
}

/** Снимаем указанную сумму с заданного счета */
public synchronized FunnyMoney withdraw(String name, String password,
     int amount)
throws RemoteException, BankingException
{
int balance = 0;
Statement s = null;
try {
 balance = verify(name, password);
 if (balance < amount)
 throw new BankingException("Недостаточно средств");
 s = db.createStatement();
 // Обновляем баланс счета
 s.executeUpdate("UPDATE accounts " +
   "SET balance = " + (balance - amount) + " " +
   "WHERE name='" + name + "' " +
   " AND password = '" + password + "'");
 // Добавляем строку к истории проводок
 s.executeUpdate("INSERT INTO " + name + "_history " +
   "VALUES ('Снято " + amount +
   " " + new Date() + "')");
 db.commit();
```

```
}
catch (SQLException e) {
 try { db.rollback(); } catch (Exception e2) {}
 throw new BankingException("Исключение SQL: " + e.getMessage() +
    ": " + e.getSQLState());
}
finally { try { s.close(); } catch (Exception e) {} }

return new FunnyMoney(amount);
}

/** Возвращаем баланс по указанному счету */
public synchronized int getBalance(String name, String password)
throws RemoteException, BankingException
{
int balance;
try {
 // Получаем баланс
 balance = verify(name, password);
 // Фиксируем транзакцию
 db.commit();
}
catch (SQLException e) {
 try { db.rollback(); } catch (Exception e2) {}
 throw new BankingException("Исключение SQL: " + e.getMessage() +
    ": " + e.getSQLState());
}
// Возвращаем баланс
return balance;
}

/** Получаем историю проводок для заданного счета */
public synchronized List getTransactionHistory(String name,
      String password)
throws RemoteException, BankingException
{
Statement s = null;
List list = new ArrayList();
try {
 // Вызываем метод verify для проверки пароля,
 // даже не интересуясь текущим балансом.
 verify(name, password);
 s = db.createStatement();
 // Запрашиваем все, что находится в таблице history
 s.executeQuery("SELECT * from " + name + "_history");
 // Получаем результаты запроса и помещаем их в список
 ResultSet r = s.getResultSet();
 while(r.next()) list.add(r.getString(1));
 // Фиксируем транзакцию
 db.commit();
}
catch (SQLException e) {
```

```
 try { db.rollback(); } catch (Exception e2) {}
 throw new BankingException("Исключение SQL: " + e.getMessage() +
    ": " + e.getSQLState());
}
finally { try { s.close(); } catch (Exception e) {} }
// Возвращаем список с историей проводок.
return list;
}

/**
* Этот метод main() является самостоятельной программой,
* которая определяет, к какой базе данных и с каким
* драйвером подключаться, соединяется с ней, создает
* объект RemoteDBBankServer и регистрирует его в реестре,
* тем самым делая его доступным для использования клиентами
**/
public static void main(String[] args) {
try {
 // Создаем новый объект Properties. Пытаемся инициализировать
 // его значениями из файла BankDB.props или файла, указанного
 // в командной строке, игнорируя при этом все ошибки.
 Properties p = new Properties();
 try { p.load(new FileInputStream(args[0])); }
 catch (Exception e) {
 try { p.load(new FileInputStream("BankDB.props")); }
 catch (Exception e2) {}
 }

 // Файл BankDB.props (или файл, указанный в командной
 // строке) должен содержать свойства “driver” и “database”,
 // а также необязательные “user” и “password”.
 String driver = p.getProperty("driver");
 String database = p.getProperty("database");
 String user = p.getProperty("user", "");
 String password = p.getProperty("password", "");

 // Загрузка класса драйвера базы данных
 Class.forName(driver);

 // Соединение с базой данных, хранящей информацию о счетах
 Connection db = DriverManager.getConnection(database,
      user, password);

 // Настраиваем базу данных, чтобы разрешить группировку множественных
 // запросов и команд обновления в атомарные транзакции.
 db.setAutoCommit(false);
 db.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);

 // Создаем серверный объект, использующий соединение
 // с нашей базой данных.
 RemoteDBBankServer bank = new RemoteDBBankServer(db);

 // Читаем системное свойство для выяснения имени сервера.
 // По умолчанию используем “SecondRemote”.
```

```
    String name = System.getProperty("bankname", "SecondRemote");

    // Регистрируем сервер под этим именем
    Naming.rebind(name, bank);

    // И сообщаем всем, что мы работаем.
    System.out.println(name + " открыт и ждет клиентов.");
   }
   catch (Exception e) {
    System.err.println(e);
    if (e instanceof SQLException)
    System.err.println("SQL State: " +
       ((SQLException)e).getSQLState());
    System.err.println("Формат: java [-Dbankname=<name>] " +
     "com.davidflanagan.examples.sql.RemoteDBBankServer " +
      "[<dbpropsfile>]");
    System.exit(1);
   }
   }
  }
```

Упражнения

17-1. Чтобы начать использовать JDBC, вам прежде всего понадобятся сервер базы данных и JDBC-драйвер для него, а также потребуется умение администрировать сервер, для того

чтобы создать базу данных. Если вы еще не являетесь опытным программистом баз данных, научиться всему этому будет для вас сложнее, чем собственно программировать с помощью JDBC. Следовательно, в качестве первого упражнения достаньте и установите сервер базы данных, если вы еще не сделали это. Достаньте и установите для него драйвер JDBC. Изучите документацию как по серверу, так и по драйверу. Изучите основы языка SQL, если вы их еще не знаете, и обратите внимание на то, какие из SQL-подмножеств и SQL-расширений поддерживаются вашим сервером и JDBC-драйвером.

17-2. Пример 17.1 является универсальной программой-интерпретатором SQL, отображающей результаты запроса базы данных в формате простейшей текстовой таблицы. Доработайте эту программу так, чтобы она выводила результаты запроса, используя синтаксис HTML-таблицы и приводя результат к форме, пригодной для показа в броузере. Протестируйте вашу программу, посылая запросы какой-нибудь существующей базе данных.

17-3. Если вы знакомы с CGI-программированием, снова измените пример 17.1 так, чтобы он мог использоваться в качестве CGI-сценария. Создайте соответствующую HTML-страницу, которая передавала бы SQL-запросы пользователя серверной Java-про-

грамме и отображала HTML-таблицу, сгенерированную вашей программой.

17-4. Напишите программу для генерации таблицы базы данных, содержащей все файлы и каталоги вашего компьютера (или, по крайней мере, все файлы и каталоги ниже указанного каталога). Каждый элемент в таблице базы данных должен содержать имя файла, размер, дату последнего изменения, булево значение, показывающее, является он файлом или каталогом. Запустите эту программу для генерации базы данных файлов. Напишите еще одну программу, которая дает пользователю возможность создавать полезные запросы для этой базы данных, такие как «вывести список всех файлов, размер которых больше 1 мегабайта, а возраст больше 1 месяца» или «вывести список всех файлов с расширением *.java*, которые были изменены сегодня». Разработайте и напишите CGI-программу, позволяющую пользователю без знания SQL посылать такие структурированные запросы.

Глава 18

Сервлеты и JSP

Сервлет – это Java-класс, реализующий интерфейс `javax.servlet.Servlet` или, что чаще встречается, расширяющий абстрактный класс `javax.servlet.http.HttpServlet`. Сервлеты и Java Servlet API являются расширением архитектуры веб-серверов.[1] Вместо обслуживания исключительно статических веб-страниц веб-сервер с поддержкой сервлетов может вызывать их методы с целью динамического создания содержимого во время исполнения. Эта модель предлагает ряд преимуществ перед традиционными CGI-скриптами. Наиболее заметным является то, что экземпляр сервлета может существовать в промежутках между клиентскими запросами, поэтому серверу не требуется постоянно порождать внешние процессы.

JSP является сокращением от JavaServer Pages (серверные страницы Java). Эта архитектура построена на базе Servlet API. Страница JSP содержит HTML-код, перемежающийся с исходным текстом на Java, специальными JSP-тегами внешних библиотек тегов. Веб-сервер с поддержкой сервлетов компилирует страницы JSP «на лету», преобразуя исходный текст JSP в сервлеты, генерирующие динамический вывод.

Эта глава содержит примеры как сервлетов, так и страниц JSP, а заканчивается примером, демонстрирующим, как сервлеты и страницы JSP могут быть интегрированы в легко развертываемое веб-приложение. Однако начинается она с описания всего необходимого для ком-

[1] На самом деле пакет `javax.servlet` может использоваться с несколькими типами серверов, реализующими протокол «запрос–ответ». Однако на сегодняшний день единственной областью широкого применения сервлетов являются веб-серверы, и в этой главе сервлеты обсуждаются исключительно в таком контексте.

пиляции, развертывания, запуска и обслуживания сервлетов. Более подробную информацию по сервлетам и JSP см. в книгах издательства O'Reilly: «Java Servlet Programming» Джейсона Хантера (Jason Hunter) и Уильяма Кроуфорда (William Crawford) и «JavaServer Pages» Ханса Бергстена (Hans Bergsten).

Настройка сервлетов

Для выполнения примеров, приведенных в этой главе, вам понадобится следующее:

- Веб-сервер для хранения примеров.
- *Контейнер сервлетов* (*servlet container*), или *машина сервлетов* (*servlet engine*), используемый веб-сервером для выполнения сервлетов. Для того чтобы иметь возможность запустить примеры с JSP, ваш контейнер сервлетов должен также иметь поддержку JSP.
- Файлы классов для Servlet API, для того чтобы вы смогли откомпилировать примеры.
- *Дескриптор развертывания* (*deployment descriptor*), указывающий контейнеру сервлетов, как сопоставить URL-адреса классам сервлетов.

Этот список выглядит гораздо страшнее, чем он есть на самом деле, что вы и увидите в следующем разделе.

Контейнер сервлетов

Точно так же, как есть множество доступных веб-серверов, есть и большой выбор контейнеров сервлетов. Я сам использовал и вам рекомендую Tomcat – продукт с открытым исходным кодом из проекта Jakarta. Собственно, Jakarta является проектом Apache Software Foundation, организации, разрабатывающей веб-сервер с открытым кодом Apache. Большой вклад в развитие сервера Tomcat внесла фирма Sun. Он является наследником Java Servlet Development Kit, который может использоваться веб-сервером Apache, а также различными коммерческими веб-серверами. Вместе с тем, в состав Tomcat входит веб-сервер, целиком написанный на Java, который вы можете запустить на своей локальной машине.

Если вы решили использовать Tomcat, вы можете загрузить его с сайта *http://jakarta.apache.org*. На момент написания книги текущей версией Tomcat была версия 3.1. К тому времени, когда вы будете читать эту главу, она, возможно, будет уже другой, поэтому убедитесь, что вы загружаете последнюю версию продукта, и прочитайте инструкции по его установке и настройке, распространяемые вместе с сервером.[1]

[1] На момент перевода книги была доступна версия Tomcat 4.0.3, поддерживающая спецификации Servlets 2.3 и JSP 1.2. Все примеры из книги работают корректно под Tomcat этой версии. – *Примеч. науч. ред.*

Для установки Tomcat просто распакуйте загруженный вами архивный файл. Для запуска Tomcat просто зайдите в каталог, в который вы его установили, и введите следующую команду:[1]

```
% bin/tomcat.sh run          # Unix/Linux
C:\> bin\tomcat.bat run     # Windows
```

По умолчанию веб-сервер Tomcat прослушивает порт 8080, поэтому после того, как вы запустили Tomcat, направьте свой броузер по адресу *http://localhost:8080/* или *http://127.0.0.1:8080/*. Если все работает правильно, вы должны увидеть начальную страницу с информацией о сервере Tomcat. Конечно, для получения подробной информации вы должны прочитать инструкции, распространяемые вместе с сервером.

Если у вас уже есть работающий веб-сервер и контейнер сервлетов, можете двигаться дальше и использовать его (после того, как внимательно изучите документацию по нему). Однако обратите внимание, что примеры этой главы написаны в соответствии со спецификациями Servlets 2.2 и JSP 1.1. Если ваш контейнер сервлетов не поддерживает эти версии спецификации, некоторые примеры могут не заработать.

Компиляция сервлетов

Ваш контейнер сервлетов должен входить в состав файла JAR, содержащего файлы классов для Servlet API. Возможно, он будет находиться в файле с именем *lib/servlets.jar*.[2] Этот файл будет вам нужен для компиляции классов, содержащихся в этой главе. Отредактируйте переменную окружения `CLASSPATH`, включив в нее данный файл, или поместите его копию в каталог *lib/jre/ext/* вашего Java SDK, где виртуальная машина Java сможет найти его автоматически. Сделав это, вы можете откомпилировать классы сервлета при помощи *javac* точно так же, как вы компилируете другие классы Java. Если вы не смогли найти классы Servlet API в вашем дистрибутиве контейнера сервлетов, вы можете загрузить их с веб-сайта Sun (*http://java.sun.com/products/servlet/*). Убедитесь в том, что вы загружаете версию, соответствующую спецификациям, поддерживаемым вашим контейнером сервлетов.

Установка и запуск сервлетов

После того как вы откомпилировали сервлет, вам нужно поместить полученный файл (или файлы) класса туда, где веб-сервер сможет найти его. В общем случае эта задача по развертыванию является спе-

---

[1] Сейчас старт/стопный скрипт называется `catalina.bat` (Windows) или `catalina.sh` (Unix). – *Примеч. науч. ред.*

[2] В последней версии Tomcat файл *servlet.jar* расположен в каталоге *common/lib*. – *Примеч. науч. ред.*

цифичной для каждого конкретного сервера, поэтому для выяснения, как это сделать, вам необходимо изучить документацию по вашему серверу.

Однако, к счастью, в версии 2.2 Servlet API было введено понятие *веб-приложения* (*web application*) – набора одного или нескольких взаимодействующих сервлетов с необходимыми им вспомогательными файлами, а также была стандартизирована методика по развертыванию сервлетов, составляющих веб-приложение. В состав всех веб-приложений входит файл с именем *WEB-INF/web.xml*, предоставляющий всю информацию, необходимую для развертывания сервлетов. Веб-приложения могут быть упакованы в архивы WAR, использующие формат архива JAR. Мы рассмотрим архивы WAR и файл *web.xml* в конце этой главы.

Онлайновая версия примеров этой главы включает и готовый WAR-файл. Если вы используете Tomcat или другой контейнер сервлетов, совместимый с версией 2.2, вы можете установить все примеры из этой главы, просто поместив WAR-файл туда, куда указано в документации по контейнеру. При использовании Tomcat 3.1 вы можете просто скинуть WAR-файл в каталог *webapps/* и перезагрузить Tomcat. (Когда Tomcat 3.1 встречает новое веб-приложение, развернутое в WAR-архиве, первым делом он извлекает все файлы из этого архива и сохраняет их в подкаталоге каталога *webapps/*. Это удобно тем, что вы сможете начать использовать готовый архив *javaexamples2.war* и легко модифицировать, перекомпилировать и переразвертывать примеры без необходимости повторного создания WAR-файла.)

При использовании WAR-архивов имя WAR-файла используется как префикс в URL для всех сервлетов и иных файлов, находящихся в архиве. Предположим, что вы разворачиваете архив *javaexamples2.war* в каталоге *webapps/* сервера Tomcat. Если этот WAR-архив содержит на корневом уровне файл с именем *index.html*, URL для этого файла будет таким:

```
http://localhost:8080/javaexamples2/index.html
```

URL для сервлетов похожи. Файл *web.xml* задает соответствие между именами сервлетов и именами реализующих их классов. Например, файл *web.xml*, находящийся в *javaexamples2.war*, связывает имя `hello` с классом `com.davidflanagan.examples.servlet.Hello`. Для запуска этого сервлета вы просто направляете ваш броузер по адресу:

```
http://localhost:8080/javaexamples2/servlet/hello
```

Когда контейнеру сервлетов Tomcat сказано запустить этот сервлет, он знает, что его класс нужно искать в WAR-архиве по имени файла:

```
WEB-INF/classes/com/davidflanagan/examples/servlet/Hello.class
```

Сервлет Hello World

В примере 18.1 приведен листинг программы *Hello.java*, реализующей простой сервлет «Hello world», показанный на рис. 18.1. Сервлет `Hello` является производным от `javax.servlet.http.HttpServlet` и замещает его метод `doGet()` для генерации вывода в ответ на запрос HTTP GET. Кроме этого, он замещает метод `doPost()`, получая возможность таким же образом отвечать и на запрос POST. Метод `doGet()` выводит строку текста в формате HTML. По умолчанию эта строка – «Hello World».

***Рис. 18.1.** Сервлет Hello*

Однако если сервлет `Hello` сможет определить имя пользователя, он поприветствует его по имени. Сервлет ищет имя пользователя в двух местах, начиная с запроса HTTP (объекта `HttpServletRequest`), как значение параметра с именем `username`. Если сервлет не может найти требуемый параметр с таким именем, он ищет связанный с данным запросом объект `HttpSession` и проверяет, имеет ли этот объект атрибут с именем `username`. Веб-сервер с поддержкой сервлетов (то есть контейнер сервлетов) предоставляет уровень отслеживания сеансов (session-tracking layer), построенный над не поддерживающим состояние протоколом HTTP. Объект `HttpSession` позволяет сервлету устанавливать и опрашивать именованные атрибуты, связанные с отдельным клиентским сеансом. Ниже в этой главе мы увидим пример, использующий возможности по отслеживанию сеансов сервлета `Hello`.

Пример 18.1. Hello.java

```
package com.davidflanagan.examples.servlet;
import javax.servlet.*;      // основные классы и интерфейсы сервлетов
import javax.servlet.http.*; // классы, относящиеся к HTTP-сервлетам
import java.io.*;            // сервлеты выполняют ввод/вывод и генерируют
исключения

/**
 * Этот простой сервлет приветствует пользователя.
 * Он просматривает объекты запроса и сеанса, пытаясь
 * поприветствовать пользователя по имени.
 **/
public class Hello extends HttpServlet {
```

```
// Этот метод вызывается, когда сервлету направляется запрос HTTP GET
 public void doGet(HttpServletRequest request,
       HttpServletResponse response) throws IOException
 {
  // Смотрим, указано ли имя пользователя в запросе
  String name = request.getParameter("username");

  // Если нет, проверяем объект сеанса. Веб-сервер или контейнер сервлетов
  // выполняют для сервлета автоматическое отслеживание сеансов
  // и связывают с каждым сеансом объект HttpSession.
  if (name == null)
   name = (String)request.getSession().getAttribute("username");

  // Если имя пользователя не было найдено ни в одном месте,
  // используем имя по умолчанию
  if (name == null) name = "World";

  // Указываем тип вывода, который мы генерируем.
  // Если данный сервлет вложен внутрь другого сервлета
  // или страницы JSP, эта установка будет игнорироваться
  response.setContentType("text/html");

  // Получаем поток, в который мы можем записывать вывод
  PrintWriter out = response.getWriter();

  // И наконец, выполняем наш вывод.
  out.println("Hello " + name + "!");
 }

 // Этот метод вызывается, когда сервлету направляется
 // запрос HTTP POST. Он вызывает метод doGet(), поэтому
 // сервлет корректно обрабатывает оба типа запросов.
 public void doPost(HttpServletRequest
   request,HttpServletResponse response) throws IOException
 {
  doGet(request, response);
 }
}
```

Запуск сервлета Hello

Перед тем как вы сможете запустить этот сервлет, вы должны откомпилировать и развернуть его так, как было описано в начале главы. Чтобы запустить сервлет, направьте ему запрос при помощи броузера. Используемый вами URL будет зависеть от того, где запущен веб-сервер и как вы развернули сервлет. Если вы используете запущенный на локальной машине Tomcat (или другой контейнер сервлетов, совместимый с версией 2.2), если этот веб-сервер прослушивает порт с номером 8080 и если ваш сервлет развернут как часть веб-приложения с именем `javaexamples2`, то вы сможете запустить сервлет, направив ваш броузер по адресу:

```
http://localhost:8080/javaexamples2/servlet/hello
```

Он должен отобразить веб-страничку, на которой написано «Hello World». Чтобы получить более сложный результат, добавим к URL требуемый параметр, такой как (он был использован для генерации вывода, показанного на рис. 18.1):

```
http://localhost:8080/javaexamples2/servlet/hello?username=David
```

Инициализация и постоянство сервлетов: сервлет Counter

В примере 18.2 приведен листинг *Counter.java* сервлета, поддерживающего любое количество именованных счетчиков. Каждый раз, когда запрашивается значение одного из счетчиков, сервлет увеличивает этот счетчик и выводит его новое значение. Данный сервлет может использоваться в качестве счетчика посещений на многих веб-страницах, но помимо этого он может подсчитывать события любого другого типа.

Сервлет определяет методы `init()` и `destroy()` и сохраняет свое состояние в файле, поэтому он не теряет значения счетчиков при выключении сервера (или контейнера сервлетов). Для того чтобы понять назначения методов `init()` и `destroy()`, вы должны кое-что узнать о жизненном цикле сервлета. Экземпляры сервлета обычно не создаются заново при каждом клиентском запросе. Вместо этого после создания и до момента уничтожения сервлет может обслужить множество запросов. Сервлет, такой как `Counter`, обычно не заканчивает работу, если только не завершает работу сам контейнер сервлетов или сервлет не переходит в неактивное состояние, и контейнер пытается освободить память с целью освобождения места под другие сервлеты.

Метод `init()` вызывается, когда контейнер в первый раз создает экземпляр сервлета, перед тем как он начнет обслуживание запросов. Первым делом метод ищет значения двух *параметров инициализации*: имя файла, содержащего сохраненное состояние, и значение целого типа, указывающее, как часто требуется сохранять состояние в этом файле. После того как метод `init()` прочитал эти параметры, он считывает значения счетчиков (используя механизм сериализации объектов) из указанного файла и становится готов к началу обслуживания запросов. Значения параметров инициализации хранятся в файле развертывания *WEB-INF/web.xml*. Перед запуском этого примера вам, возможно, потребуется отредактировать этот файл для того, чтобы сообщить сервлету, где нужно сохранять состояние. Взгляните на эти строки:

```
<init-param>
 <param-name>countfile</param-name>          <!-где сохранять состояние-->
 <param-value>/tmp/counts.ser</param-value> <!-подстройте под вашу систему-->
</init-param>
```

Если имя файла */tmp/counts.ser* не применимо к вашей системе, замените его другим.

Метод `destroy()` составляет пару методу `init()`. Он вызывается после того, как сервлет завершает работу и больше не будет обслуживать клиентские запросы. Сервлет `Counter` использует этот метод для сохранения своего состояния, поэтому оно может быть корректно восстановлено после того, как контейнер сервлетов запустит сервлет снова. Однако заметьте, что метод `destroy()` вызывается, только если сервлет завершает работу штатным образом. В случае сбоя сервера или отключения питания у сервлета нет возможности сохранить состояние. Поэтому сервлет периодически сохраняет свое состояние и в методе `doGet()`, в результате чего он никогда не теряет много данных.

Метод `doGet()` в первую очередь должен определить имя счетчика, значение которого будет отображаться. Так как сервлет `Counter` предназначен для использования в различных ситуациях, он использует три способа получения имени счетчика. Во-первых, он проверяет параметр, переданный в запросе HTTP. Затем он проверяет требуемый атрибут, являющийся именованным значением, которое было связано с данным запросом контейнером сервлетов или другим сервлетом, вызвавшим сервлет `Counter` (вы увидите пример этого далее в текущей главе), и наконец, если сервлет не обнаружил имени счетчика ни одним из приведенных выше способов, он использует в качестве имени счетчика URL, через который он был вызван. Как вы увидите, контейнер сервлетов может быть сконфигурирован так, чтобы вызывать сервлет в ответ на любой URL, имеющий суффикс *.count*.

Обратите внимание, что в методе `doGet()` часть кода защищена блоком `synchronized`. Servlet API позволяет выполнять тело метода `doGet()` одновременно нескольким потокам исполнения. Даже в том случае, если будут запущены несколько потоков исполнения, представляющих несколько разных пользовательских сеансов, все они будут использовать единственную структуру данных – хеш-таблицу именованных счетчиков. Блок `synchronized` предотвращает доступ (и, возможно, порчу) нескольких потоков исполнения к совместно используемой структуре данных. В заключение отметим использование метода `log()`. При запросе начать отсчет с именем счетчика, которое раньше не использовалось, сервлет использует этот метод для внесения постоянной записи об этом событии в журнал (log-файл).

Пример 18.2. Counter.java

```
package com.davidflanagan.examples.servlet;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

/**
 * Этот сервлет поддерживает произвольный набор переменных-счетчиков,
```

```
 * увеличивая и отображая при каждом своем вызове значение одного
 * из именованных счетчиков. Он сохраняет состояния счетчиков в файле
 * на диске, поэтому их значения не теряются при выключении сервера.
 * Он может использоваться для подсчета количества посещений страницы
 * или других типов событий. Обычно не вызывается непосредственно,
 * а включается в состав других страниц при помощи технологий JSP,
 * SSI или RequestDispatcher
 **/
public class Counter extends HttpServlet {
 HashMap counts;     // Хеш-таблица: сопоставляет имена счетчиков со счетчиками
 File countfile;     // Файл, в котором сохраняются значения счетчиков
 long saveInterval;  // Как часто (в мс) сохранять состояния во время работы?
 long lastSaveTime;  // Когда мы последний раз сохраняли состояние?

 // Этот метод вызывается, когда веб-сервер первый раз
 // создает экземпляр данного сервлета. Он считывает
 // параметры инициализации, которые задаются во время
 // развертывания в файле web.xml, и загружает начальные
 // значения переменных-счетчиков из файла.
 public void init() throws ServletException {
  ServletConfig config = getServletConfig();
  try {
   //Получаем сохраненный файл
   countfile = new File(config.getInitParameter("countfile"));
  // Как часто нужно сохранять состояние во время работы?
   saveInterval =
  Integer.parseInt(config.getInitParameter("saveInterval"));
   // Состояние не могло измениться перед этим
   lastSaveTime = System.currentTimeMillis();
   // Считываем значения счетчиков
   loadState();
  }
  catch(Exception e) {
  // Если происходит ошибка, генерируем повторное исключение
  throw new ServletException("Невозможно инициализировать сервлет Counter: "
     + e.getMessage(), e);
  }
 }

 // Этот метод вызывается, когда веб-сервер останавливает
 // сервлет (что случается, когда веб-сервер выключается
 // или когда сервлет не используется). В этом методе значения
 // счетчиков сохраняются в файле, поэтому они могут быть
 // восстановлены при возобновлении работы сервлета.
 public void destroy() {
 try { saveState(); }   // Пытаемся сохранить состояние
 catch(Exception e) {}  // Игнорируем все проблемы: это лучшее, что мы можем
 }

 // Эти константы определяют имена параметра и атрибута
 // запроса, используемые сервлетом для поиска имени
 // счетчика, подлежащего увеличению.
```

```
public static final String PARAMETER_NAME = "counter";
public static final String ATTRIBUTE_NAME =
"com.davidflanagan.examples.servlet.Counter.counter";

/**
* Этот метод вызывается при обращении к сервлету.
* Он ищет параметр запроса с именем "counter" и использует
* его значение в качестве имени наращиваемой переменной-счетчика.
* Если он не находит параметр в запросе, то в качестве имени счетчика
* используется URL запроса. Это полезно, когда сервлет сопоставлен
* с суффиксом URL. Кроме этого, метод проверяет, сколько времени прошло
* с момента последнего сохранения состояния, и при необходимости снова
* сохраняет его. Это предотвращает потерю большого количества данных,
* если происходит сбой или выключение сервера
* без вызова метода destroy().
**/
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws IOException
{
// Получаем имя счетчика как параметр запроса
String counterName = request.getParameter(PARAMETER_NAME);

// Если мы не нашли его там, проверяем, передан ли он нам
// в атрибуте запроса, что имеет место, когда вывод нашего
// сервлета включается в другой сервлет
 if (counterName == null)
 counterName = (String) request.getAttribute(ATTRIBUTE__NAME);

// Если отсутствуют и параметр, и атрибут, используем URL запроса.
if (counterName == null) counterName = request.getRequestURI();

Integer count; // Это текущий счетчик

// Этот блок синхронизируется, из-за того что от разных потоков исполнения
// в одно и то же время может поступить несколько запросов. Синхронизация
// предотвращает одновременное обновление ими хеш-таблицы счетчиков.
synchronized(counts) {
 // Получаем из таблицы значение счетчика
 count = (Integer)counts.get(counterName);

 // Увеличиваем счетчик или, если он новый, вносим запись
 // в журнал событий и устанавливаем счетчик в 1.
 if (count != null) count = new Integer(count.intValue() + 1);
 else {
 // Если это счетчик, который мы раньше не использовали,
 // посылаем сообщение в журнал событий, просто для того,
 // чтобы следить за тем, что мы считаем
 log("Начинаем счет с новым счетчиком: " + counterName);
 // Начинаем считать с 1!
 count = new Integer(1);
 }

 // Сохраняем значение увеличенного (или нового) счетчика в хеш-таблице
 counts.put(counterName, count);
```

```
 // Проверяем, прошло ли saveInterval миллисекунд с момента
 // последнего сохранения нашего состояния. Если прошло –
 // Сохраняем его снова. Это предотвращает потерю данных,
 // накопленных более чем за saveInterval миллисекунд,
 // даже если сервер неожиданно даст сбой.
 if (System.currentTimeMillis() - lastSaveTime > saveInterval) {
 saveState();
 lastSaveTime = System.currentTimeMillis();
 }
} // Конец блока синхронизации

// Наконец, выводим значение счетчика. Из-за того что
// обычно данный сервлет вложен внутрь другого сервлета,
// нам не нужно беспокоиться об установке типа содержимого.
PrintWriter out = response.getWriter();
out.print(count);
}

// Метод doPost просто вызывает doGet, так как сервлет
// может быть частью страницы, загружаемой через запрос POST.
public void doPost(HttpServletRequest request,HttpServletResponse response)
throws IOException
{
doGet(request, response);
}

// Сохраняем состояние счетчиков путем сериализации хеш-таблицы
// в файл, указанный в параметрах инициализации.
void saveState() throws IOException {
ObjectOutputStream out = new ObjectOutputStream(
  new BufferedOutputStream(new FileOutputStream(countfile)));
out.writeObject(counts); // Записываем хеш-таблицу в поток
out.close();  //Никогда не забывайте закрывать свои файлы!
}

// Загружаем начальное состояние счетчиков путем десериализации
// хеш-таблицы из файла, указанного в параметрах инициализации.
// Если этот файл еще не существует, начинаем с пустой хеш-таблицы.
void loadState() throws IOException {
if (!countfile.exists()) {
 counts = new HashMap();
 return;
}
ObjectInputStream in = null;
try {
 in = new ObjectInputStream(
  new BufferedInputStream(new FileInputStream(countfile)));
 counts = (HashMap) in.readObject();
}
catch(ClassNotFoundException e) {
 throw new IOException("Файл счетчиков содержит неверные данные: " +
    e.getMessage());
}
```

```
 finally {
  try { in.close(); }
  catch (Exception e) {}
 }
 }
}
```

Запуск сервлета Counter

Наш файл *web.xml* связывает имя `counter` с сервлетом `Counter`. Для поиска имени счетчика, подлежащего наращиванию и отображению, сервлет считывает параметр запроса, также имеющий имя `counter`. Поэтому вы можете протестировать сервлет, направив свой броузер по адресу следующего вида:

```
http://localhost:8080/javaexamples2/servlet/counter?counter=numhits
```

В файле *web.xml* также указано, что сервлет `Counter` должен вызываться во всех случаях, когда у веб-сервера запрашивается любой файл, заканчивающийся суффиксом *.count* (и начинающийся именем веб-приложения). Далее в этой главе вы увидите, как *web.xml* определяет это. В данном случае в качестве имени переменной-счетчика сервлет `Counter` использует сам URL. Протестируйте этот вариант, введя следующий URL:

```
http://localhost:8080/javaexamples2/index.html.count
```

К этому моменту вы, вероятно, уже заметили, что сервлет `Counter` не выводит ничего интересного: он просто отображает число. Данный сервлет не очень полезен, когда используется сам по себе. Напротив, он предназначен для генерации вывода, включенного внутрь вывода других сервлетов. (Мы рассмотрим пару способов сделать это ниже в этой главе.) Если ваш сервер поддерживает серверные включения (server-side includes, SSI) и тег `<servlet>`, вы можете использовать эту возможность для включения вывода сервлета `Counter` в веб-страницы. (Сервер Tomcat не поддерживает SSI, вероятно, потому, что технология JSP делает SSI во многом излишней и ненужной.[1]) Для того чтобы сделать это, вы должны создать файл *test.shtml*, содержащий текст наподобие следующего:

```
<p>This page has been accessed
  <servlet name='/servlet/counter'>
    <param name='counter' value='test.shtml'>
  </servlet>
times.
```

[1] Доступная сейчас версия Tomcat 4.0.3 поддерживает SSI. – *Примеч. науч. ред.*

Доступ к базам данных из сервлетов

Одним из наиболее частых применений сервлетов (и веб-приложений вообще) является реализация программного обеспечения промежуточного уровня, расположенного между клиентом и серверами баз данных. В примере 18.3 приведен листинг *Query.java* сервлета, принимающего вводимый пользователем SQL-запрос (с помощью HTML-формы), выполняющего этот запрос (используя JDBC), а затем отображающего его результат (используя таблицу HTML). На рис. 18.2 показан результат работы этого сервлета.

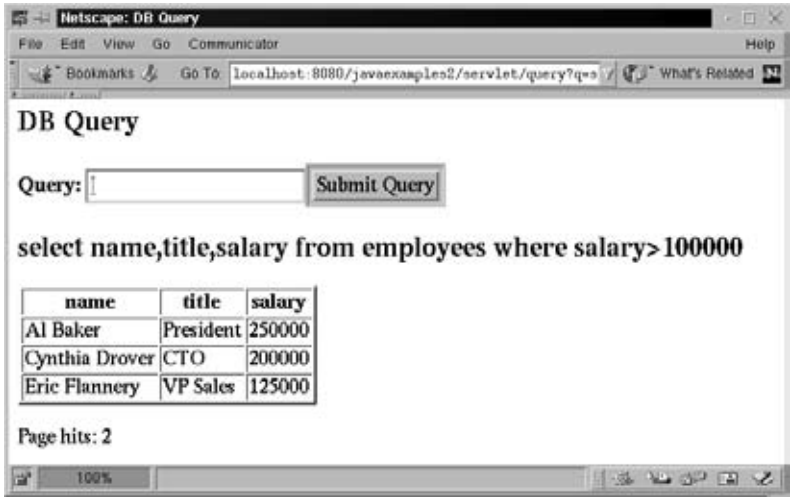


***Рис. 18.2.** Вывод сервлета Query*

Код этого сервлета достаточно понятен. Метод `init()` считывает параметры инициализации, указывающие JDBC-драйвер, который необходимо использовать, базу данных для подключения и имя пользователя с паролем, используемые для этого подключения. Метод `init()` использует эту информацию для установки соединения с базой данных, которое поддерживается в течение всей жизни сервлета. Это соединение закрывается в методе `destroy()`. Создание постоянного соединения позволяет избежать ненужных расходов на установление и закрытие соединения при каждом запросе сервлета. Однако это подразумевает, что используемый вами JDBC-драйвер является потокобезопасным и работает только при условии, что вы ограничитесь запросами базы данных и простыми командами обновления, не требующими транзакций. Если вам необходимо работать с транзакциями, вы можете засинхронизировать метод `doGet()` для гарантии того, что в каждый момент времени сервлет обрабатывает только один запрос. Или, если ваш веб-сайт имеет высокий трафик, вы можете использовать методику, известную как

пул соединений (*connection pooling*) и состоящую в создании соединений заранее и выделении их по необходимости запрашиваемым потокам исполнения. Эта и другие связанные с ней методики подробно описаны в книге Джейсона Хантера «Java Servlet Programming».

В методе `doGet()` для отправки запросов серверу базы данных и вывода результатов в виде HTML-таблицы используется стандартный для JDBC код. (Если вы не знакомы с тегом `<table>` и другими связанными с ним тегами, см. справочник O'Reilly «HTML & XHTML. The Definitive Guide» Чака Муссиано (Chuck Musciano) и Билла Кеннеди (Bill Kennedy).[1]) Кроме этого, метод `doGet()` отображает форму, позволяющую пользователю ввести SQL-запрос. Когда пользователь отсылает эту форму, броузер повторно загружает сервлет, передавая SQL-запрос пользователя в виде параметра запроса. Включение в один сервлет кода, который и создает, и обрабатывает формы, является широко используемой и полезной технологией в программировании сервлетов.

Наконец, обратите внимание, что метод `doGet()` использует объект `RequestDispatcher()`, позволяющий вызвать объект `Counter` и вставить его вывод в собственный. Для указания имени счетчика в качестве атрибута запроса в методе `doGet()` используется метод `setAttribute()`. Сервлет `Counter` содержит соответствующий вызов `getAttribute()`.

Пример 18.3. Query.java

```
package com.davidflanagan.examples.servlet;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;
import java.io.*;

/**
 * Этот класс демонстрирует, как JDBC может использоваться в сервлетах.
 * Он использует параметры инициализации (извлекаемые из файла конфигурации
 * web.xml) для создания единственного JDBC-соединения с базой данных,
 * которое совместно используется клиентами сервлета.
 ***/
public class Query extends HttpServlet {
    Connection db;     // Это общее JDBC-соединение с базой данных

    public void init() throws ServletException {
        // Считываем параметры инициализации из файла web.xml
        ServletConfig config = getServletConfig();
        String driverClassName = config.getInitParameter("driverClassName");
        String url = config.getInitParameter("url");
        String username = config.getInitParameter("username");
        String password = config.getInitParameter("password");

  // Используем эти параметры для установки соединения с базой данных
 // при возникновении ошибки, записываем ее и повторно генерируем исключения.
```

[1] Чак Муссиано и Билл Кеннеди «HTML и XHTML. Подробное руководство», 4-е издание, Символ-Плюс, 2002.

```
 try {
     Class.forName(driverClassName);
     db = DriverManager.getConnection(url, username, password);
 }
 catch (Exception e) {
     log("Невозможно создать соединение с базой данных", e);
     throw new ServletException("Запрос: невозможно инициализировать: " +
                                e.getMessage(), e);
 }
}

/** Закрываем соединение с базой данных при выгрузке сервлета */
public void destroy() {
   try { db.close(); }       // Пытаемся закрыть соединение
   catch (SQLException e) {} // Игнорируем ошибки, по крайней мере, пытаемся!
}

public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
{
    response.setContentType("text/html");   // Мы выводим HTML
    PrintWriter out = response.getWriter(); // Куда его выводить

   // Выводим заголовок документа и форму для ввода SQL-запросов.
   // Когда эта форма отсылается - сервлет загружается повторно
   out.println("<head><title>DB Query</title></head>\n" +
               "<body bgcolor=white><h1>DB Query</h1>\n" +
               "<form><b>Query: </b><input name='q'>" +
               "<input type=submit></form>");

   // Смотрим, была ли указана SQL-команда в этом запросе.
   String query = request.getParameter("q");
   if (query != null) {
   // Отображаем текст этой команды в виде заголовка страницы
   out.println("<h1>" + query + "</h1>");

   // Теперь пытаемся выполнить запрос и отображаем результаты в таблице
   Statement statement = null; // Объект для выполнения запроса
   try {
       // Создаем объект statement
       statement = db.createStatement();
      // Используем его для выполнения заданного запроса и получаем результат
      ResultSet results = statement.executeQuery(query);
      // Запрашиваем дополнительную информацию о результатах
      ResultSetMetaData metadata = results.getMetaData();
      // Сколько полей содержится в результатах?
      int numcols = metadata.getColumnCount();

      // Выводим таблицу начиная со строки заголовка, содержащей имена полей
      out.println("<table border=2><tr>");
      for(int i = 0; i < numcols; i++)
          out.print("<th>" + metadata.getColumnLabel(i+1) + "</th>");
      out.println("</tr>");

// Выполняем цикл по всем строкам итогового набора
```

```
 while(results.next()) {
     // Выводим значения всех полей для каждой строки
     out.print("<tr>");
     for(int i = 0; i < numcols; i++)
         out.print("<td>" + results.getObject(i+1) + "</td>");
     out.println("</tr>");
 }
 out.println("</table>"); // Конец таблицы
}
catch (SQLException e) {
   // Если возникает ошибка (как правило, ошибка SQL) -
   // показываем ее пользователю, чтобы он смог исправить ее
   out.println("Ошибка SQL: " + e.getMessage());
}
finally { // Что бы ни произошло, всегда закрываем объект Statement
    try { statement.close(); }
    catch(Exception e) {}
}
}

 // Теперь отображаем количество обращений к данной странице,
 // вызывая сервлет Counter и вставляя его вывод в нашу страницу.
 // Это выполняется при помощи объекта Request Dispatcher
 RequestDispatcher dispatcher =
     request.getRequestDispatcher("/servlet/counter");
 if (dispatcher != null) {
    out.println("<br> Количество посещений страницы:");
   // Добавляем атрибут запроса, сообщающий сервлету, что нужно подсчитывать.
   // Используем имя атрибута, определенного в сервлете Counter, и используем
   // имя этого класса в качестве уникального имени счетчика.
   request.setAttribute(Counter.ATTRIBUTE_NAME,Query.class.getName());
   // Указываем диспетчеру вызвать этот сервлет и включить его вывод
   dispatcher.include(request, response);
}

 // Наконец, завершаем вывод HTML
 out.println("</body>");
}
}
```

Запуск сервлета Query

Для того чтобы самим запустить сервлет `Query`, вам потребуется отредактировать файл *WEB-INF/web.xml* и установить значения различных параметров инициализации, которые укажут ему, какой JDBC-драйвер использовать, с какой базой данных соединиться и т. д. Ваш контейнер сервлетов должен уметь находить классы JDBC-драйверов. Вам следует проверить это, поместив JAR-файл с классами драйверов в каталог *WEB-INF/lib/* веб-приложения, но, к сожалению, Tomcat 3.1 не сможет найти драйвер базы данных в каталоге *lib/*.[1] (Однако он мо-

[1] В версии Tomcat 4.0.3 эта ошибка устранена. – *Примеч. науч. ред.*

жет находить хранящиеся здесь классы сервлетов, поэтому, возможно, эта ошибка как-то связана со способом загрузки драйверов баз данных при помощи метода `Class.forName()`.) В любом случае решение состоит во включении JAR-архива с вашим JDBC-драйвером в переменную `CLASSPATH` перед запуском Tomcat.

JSP–форма входа в систему

Давайте еще раз посмотрим на метод `doGet()`, показанный в примере 18.3. Он содержит несколько вызовов методов `println()`, выводящих теги HTML. Одна из проблем сервлетов такого типа состоит в том, что в них жестко включаются теги HTML, спрятанные внутри классов Java, куда не могут добраться ни веб-дизайнеры, ни художники-оформители. Серверные страницы Java, или JSP (JavaServer Pages), представляют собой попытку исправить эту ситуацию. Вместо встраивания HTML-тегов в Java-код, страницы JSP встраивают Java-код в HTML-страницы.

Пример 18.4 содержит листинг страницы JSP *login.jsp*. Эта страница отображает окно входа в систему, показанное на рис. 18.3, и выполняет простейшую (и небезопасную) проверку пароля перед перенаправлением броузера пользователя к некоторой другой странице (указанной через параметр запроса).

Первое, что вы заметите в этом примере JSP, – это то, что он содержит много тегов в стиле HTML, начинающихся с `<%` и заканчивающихся `%>`. Это JSP-теги. В следующей таблице приведен краткий обзор JSP-тегов и их назначения.[1]

Тег	Назначение
`<%--...--%>`	Комментарий JSP. В отличие от комментариев HTML, JSP-комментарии удаляются в процессе компиляции и никогда не появляются в итоговой странице
`<%@page...%>`	Директива `@page`. Должна содержаться в каждой JSP-странице. Ее атрибуты задают язык страницы, тип содержимого, размер буфера страницы и список импортируемых пакетов
`<%@include file="URL"%>`	Включает указанный файл во время компиляции

[1] Заметьте, что здесь я не пытаюсь дать полную информацию по синтаксису JSP. С этой целью обратитесь к справочникам по JSP, таким как книга «JavaServer Pages», O'Reilly. Также вы можете найти документацию и учебные материалы на веб-сайте Sun (*http://java.sun.com/products/jsp*). Одним из наиболее полезных источников является «Java Syntax Reference Card» на *http://java.sun.com/products/jsp/pdf/card11a.pdf*.

Тег	Назначение
`<%@taglib uri="taglibId"prefix="tagPrefix"%>`	Объявляет библиотеку тегов для данной страницы. Атрибут `uri` уникально идентифицирует библиотеку, а атрибут `prefix` указывает префикс, под которым она будет известна на странице
`<%!...%>`	Тег объявлений. В эти разделители заключается Java-код, который становится методами и полями итогового класса сервлета
`<%=...%>`	Тег выражений. Содержит выражения Java. Во время выполнения тег заменяется значением этого выражения
`<%...%>`	*Скриптлет*. В эти разделители заключается Java-код, который становится частью метода `jspService()`, являющегося JSP-версией методов `goGet()` и `doPost()`

Вы могли бы заметить, что в скриптлетах и выражениях Java используются несколько локальных переменных, автоматически объявляемых системой JSP. В следующей таблице показаны доступные переменные.

Имя	Тип
`request`	`javax.servlet.http.HttpServletRequest`
`response`	`javax.servlet.http.HttpServletResponse`
`session`	`javax.servlet.http.HttpSession`
`config`	`javax.servlet.ServletConfig`
`application`	`javax.servlet.ServletContext`
`out`	`javax.servlet.jsp.JspWriter`

Рис. 18.3. *Страница login.jsp*

Пример 18.4 начинается с директив `@page` и `@taglib`. За ними следует тег объявлений, определяющий метод для простой проверки пароля.[1] За объявлением следует тег скриптлетов, который содержит Java-код, выполняющийся при каждом обращении к странице JSP. Этот код извлекает из только что принятой формы имя и пароль входа в систему (если только это не первый запрос страницы) и пытается сверить их. Если проверка завершилась положительно, скриптлет запоминает имя пользователя в объекте сеанса сервлета и отсылает команду перенаправления броузеру пользователя, что заставляет его загрузить страницу, которая была указана в параметре запроса `nextpage`.

Если проверка пароля не была выполнена успешно, тег скриптлета заканчивается и далее следует блок HTML-текста. HTML-текст создает форму входа в систему, позволяющую пользователю ввести свое имя и пароль. Теги HTML произвольно чередуются с тегами `<%=...%>`, которые содержат выражения Java, значения которых будут подставлены в HTML-вывод. Кроме этого в HTML-коде содержатся пользовательские теги `<decor:box>` и `</decor:box>`. Это вызовы тега `box` из библиотеки `decor`, импортируемой вверху страницы директивой `<%@taglib...%>`. Этот пользовательский тег создает прямоугольник с рамкой и заголовком, показанный на рис. 18.3. Далее в этой главе (в примере 18.9) вы увидите, как реализуются эти теги и делаются доступными для JSP-страниц. Заметим, что в JSP-страницах может также использоваться множество тегов с префиксом `jsp:`. Использование этих тегов не требует указания директивы `@taglib`. В этом примере их нет, но в следующих двух они используются.

Пример 18.4. login.jsp

```
<%--login.jsp
 Следующие две строки являются JSP-директивами. Директива @page
 сообщает компилятору JSP, что данная страница содержит Java-код
 (а не JavaScript, например) и генерирует HTML (а не XML, например).
 Вторая директива сообщает компилятору JSP, что страница использует
 библиотеку тегов с заданным уникальным идентификатором,
 теги которой предваряются префиксом "decor" (только на этой странице).
--%>
<%@page language='java' contentType='text/html'%>
<%@taglib uri='http://www.davidflanagan.com/tlds/decor_0_1.tld'
 prefix='decor'%>

<%--
 Код, расположенный ниже между разделителями <%!...%>, является блоком
 объявлений. Когда эта JSP-страница компилируется в сервлет, этот код
 используется для определения членов класса Servlet.
--%>
```

[1] Метод `verify()` включен сюда с целью продемонстрировать, как методы Java могут включаться в JSP-страницы. В реальной программе этот метод следует выносить во внешний класс, независимый от конкретной страницы.

```
<%! // Начало блока объявлений

// Этот метод выполняет простую проверку пароля. В настоящем
// приложении этот метод, возможно, будет сверяться с базой данных паролей.
public boolean verify(String username, String password) {
 // Принимаем любое имя пользователя, если пароль – "java"
 return ((username!=null) && (password!=null) && password.equals("java"));
}

%> <%-- Конец блока объявлений --%>

<%--
 Следующий блок кода расположен между <% и %>, которые
 помечают его как скриптлет. После компиляции JSP-страницы
 этот код становится частью метода service() сервлета.
 Блоки скриптлетов перемежаются с HTML-тегами, которые тоже
 компилируются в метод service() и выводятся сервлетом один
 к одному. Обратите внимание, что этот скриптлет
 заканчивается в середине блока else{}, и этот блок
 закрывается уже в следующем скриптлете, размещенном далее.
--%>

<% // Начало скриптлета
// Этот параметр запроса является обязательным. Он указывает,
// что нужно отображать, если попытка входа в систему закончится успешно
String nextPage = request.getParameter("nextpage");

// В этом параметре запроса указан заголовок для формы входа в систему
String title = request.getParameter("title");
if (title == null) title = "Please Log In"; // Если не указан, используем
                                            // по умолчанию

// Ищем в запросе имя пользователя и пароль
String username = request.getParameter("username");
String password = request.getParameter("password");

// Если имя пользователя и пароль определены и корректны, сохраняем имя
// в сеансе и выполняем перенаправление на указанную страницу.
// Мы делаем это, не выводя сами никакого содержимого.
if ((username != null) && (password != null) && verify(username, password)) {
 session.setAttribute("username", username);
 response.sendRedirect(nextPage);
}
else {
 // В противном случае мы должны вывести форму входа в систему.
 // Если оба свойства для имени пользователя и пароля
 // не определены, значит, мы делаем это в первый раз, и нам
 // нужно отобразить только форму. Если же они определены,
 // мы имеем ошибку при попытке входа в систему, поэтому
 // дополнительно показываем сообщение "please try again".
 String message = "";
 if ((username != null) || (password != null)) {
 message = " Неправильное имя пользователя или пароль.
             Пожалуйста, попробуйте еще раз.";
```

```
 }
%>

<%-- Это тело блока else, начатого выше. В нем выводится
 страница входа в систему. Это чистый HTML с небольшим
 количеством выражений Java, заключенных в теги <%= %>.
 Здесь также содержатся теги из пользовательской библиотеки
 тегов, рассматриваемой ниже
--%>
 <head><title>Login</title></head>
 <body bgcolor='white'>
 <br><br><br> <%-- Небольшой отступ от верха страницы --%>
 <%--Пользовательский тег: вводим декоративную рамку --%>
 <decor:box color='yellow' margin='25' borderWidth='3' title='Login'>
 <div align=center> <%-- Центрируем находящееся внутри рамки --%>
 <%-- Выводим заголовок формы входа в систему и необязательное
      сообщение об ошибке --%>
 <font face='helvetica'><h1><%=title%></h1></font>
 <font face='helvetica' color='red'><b><%=message%></b></font>
 <%-- Теперь отображаем HTML-форму, куда пользователь может
      вводить информацию для входа в систему --%>
 <form action='login.jsp' method='post'>
 <table> <%-- Используем таблицу для выравнивания формы входа в систему --%>
 <tr> <%--Первая строка: имя пользователя --%>
 <td align='right'>
 <b><font face='helvetica'>Username:</font></b>
 </td>
 <td><input name='username'></td>
 </tr><tr> <%--Вторая строка: пароль --%>
 <td align='right'>
 <b><font face='helvetica'>Password:</font></b>
 </td>
 <td><input type='password' name='password'></td>
 </tr><tr> <%-- Третья строка: кнопка login --%>
 <td align='center' colspan=2><font face='helvetica'><b>
 <input type=submit value='Login'>
 </b></font></td>
 </tr>
 </table>
 <%-- Форма должна также содержать несколько скрытых полей,
      чтобы страница смогла вернуть себе параметры nextpage и title --%>
 <input type='hidden' name='nextpage' value='<%=nextPage%>'>
 <input type='hidden' name='title' value='<%=title%>'>
 </form>
 </div>
 </decor:box> <%-- Конец пользовательского тега --%>
 </body> <%-- Конец блока HTML --%>
<%
} // Это последний скриптлет, закрывающий блок else,
  // начатый выше.
%>
```

Запуск login.jsp

JSP-файлы не требуют компиляции перед их запуском. Контейнер JSP автоматически выполняет компиляцию при первом запросе JSP-страницы и каждый раз после ее очередной модификации. В отличие от файлов классов сервлетов, которые должны находиться в каталоге *WEB-INF/classes*, страницы JSP обрабатываются точно так же, как страницы HTML и файлы изображений, и могут находиться в любом месте иерархии видимых пользователю файлов приложения. В файле WAR примера страница *login.jsp* размещена в каталоге верхнего уровня и доступна через URL наподобие следующего:

```
http://localhost:8080/javaexamples2/login.jsp
```

Вы можете заметить задержку, когда вы запрашиваете этот URL в первый раз. В это время контейнер JSP компилирует JSP-файл в сервлет.

Для корректной работы *login.jsp* ей необходим параметр запроса `nextpage`, чтобы она могла перенаправить куда-нибудь броузер пользователя после успешного входа в систему. Страница *login.jsp* сохраняет указанное при входе в систему имя пользователя в объекте сеанса под именем `username` в том же самом месте, где его ищет сервлет `Hello`. Таким образом, вы можете использовать эти сервлеты совместно при помощи URL, который выглядит так:

```
http://localhost:8080/javaexamples2/login.jsp?nextpage=servlet/hello
```

Сделав это, вы предоставите сервлету `Hello` имя пользователя, но не обеспечите никакого действительного управления доступом. Пользователь может запустить `Hello` непосредственно через URL */servlet/hello*. В следующем разделе мы увидим способ сделать вход в систему обязательным.

Передача запросов

В конце примера 18.3 мы видели пример использования класса `RequestDispatcher`, позволяющего вставить вывод одного сервлета в вывод другого. Кроме метода `include()` в классе `RequestDispatcher` есть метод `forward()`, который передает (transfers), или направляет (forwards), запрос от одного сервлета другому. При вызове этого метода обработка первого сервлета приостанавливается, и работу по генерации вывода для пользователя берет на себя второй сервлет. Единственное ограничение при передаче запросов сервлетам состоит в том, что до вызова второго сервлета первый не должен начинать генерировать вывод. Сервлет, который может передать вызов другому сервлету, обычно буферизирует свой вывод до окончания обработки запроса другой страницей. JSP-страницы выполняют буферизацию автоматически и позволяют задать размер буфера в директиве `@page`.

JSP представляет интерфейс к классу `RequestDispatcher` через два пользовательских тега, которые являются частью встроенной библиотеки тегов JSP. (Эта библиотека тегов всегда доступна через префикс `jsp:`; ее не нужно объявлять в директиве `@taglib`.) Тег `<jsp:include>` выполняет вставку сервлета (или страницы JSP), а тег `<jsp:forward>` делает перенаправление. В примере 18.5 приведен листинг *forcelogin.jsp*, который является фрагментом страницы JSP, демонстрирующим использование тега `<jsp:forward>`.

forcelogin.jsp не является законченной страницей. Это фрагмент страницы, предназначенный для статического включения в другую JSP-страницу при помощи директивы `@include`:

```
<%@include file="forcelogin.jsp"%>
```

Важно понять, что директива `@include` выполняет статическое включение во время компиляции, а не динамическое включение во время запроса, выполняемое тегом `<jsp:include>`. Пример 18.6 включает *forcelogin.jsp* с помощью директивы `@include` точно таким же способом.

Код страницы *forcelogin.jsp* проверяет, определен ли в объекте сеанса атрибут `username`. Если нет, он делает вывод, что пользователь не вошел в систему, и использует тег `<jsp:forward>` для передачи запроса пользователя на страницу *login.jsp*, которую мы видели в примере 18.4. Кроме этого, он использует тег `<jsp:param>` для задания значения параметру запроса `nextpage` с целью сообщить *login.jsp*, на какую страницу возвращаться после завершения входа в систему. Наконец, из-за того что *forcelogin.jsp* не является самостоятельной JSP-страницей, у нее нет собственного URL, поэтому для выяснения URL страницы, в которую она была вложена, в ней используется метод `getRequestURI()`.

Пример 18.5. forcelogin.jsp

```
<%--forcelogin.jsp
 Это фрагмент JSP-кода. Он не является самостоятельной страницей,
 а предназначен для включения в другие JSP-страницы. Он ищет в объекте
 сеанса атрибут "username", а если не находит его, направляет запрос
 на страницу login.jsp, которая выполняет аутентификацию пользователя
 и сохраняет его имя в объекте сеанса. Этот код посылает странице login.jsp
 параметр запроса nextpage, для того чтобы эта страница знала, куда нужно
 возвращаться после успешного входа пользователя в систему.
 Обратите внимание, что параметры запроса, переданные на страницу входа
 в систему, не сохраняются, когда она выполняет перенаправление
 обратно на эту страницу.
--%>
<%
// Проверяем, определен ли уже в объекте Session атрибут username
String _username = (String) session.getAttribute("username");
if (_username == null) { // Если пользователь еще не вошел в систему
 String _page = request.getRequestURI();
%>
<%-- Вызываем страницу login.jsp. Тег <jsp:forward> вызывает страницу
```

```
 непосредственно на сервере без отправки перенаправления клиенту.
 Теги <jsp:param> задают параметры запроса,
 передаваемые странице login.jsp --%>
<jsp:forward page='login.jsp'>
   <jsp:param name='nextpage' value='<%=_page%>'/>
   <jsp:param name='title' value='<%= "You must log in to access " + _page%>'/>
</jsp:forward>
<%}%>
```

Страницы JSP и JavaBeans

В примере 18.6 показан листинг JSP-страницы *portal.jsp*. Этот пример объединяет несколько сервлетов и технологий программирования сервлетов, продемонстрированных в текущей главе, с целью создания простейшего интернет-портала, отображающего для зарегистрированных пользователей сконфигурированную им страницу. Она показана на рис. 18.4.

Главным новшеством в примере 18.6 является использование компонентов JavaBeans (см. пример 18.7) для выделения функциональных

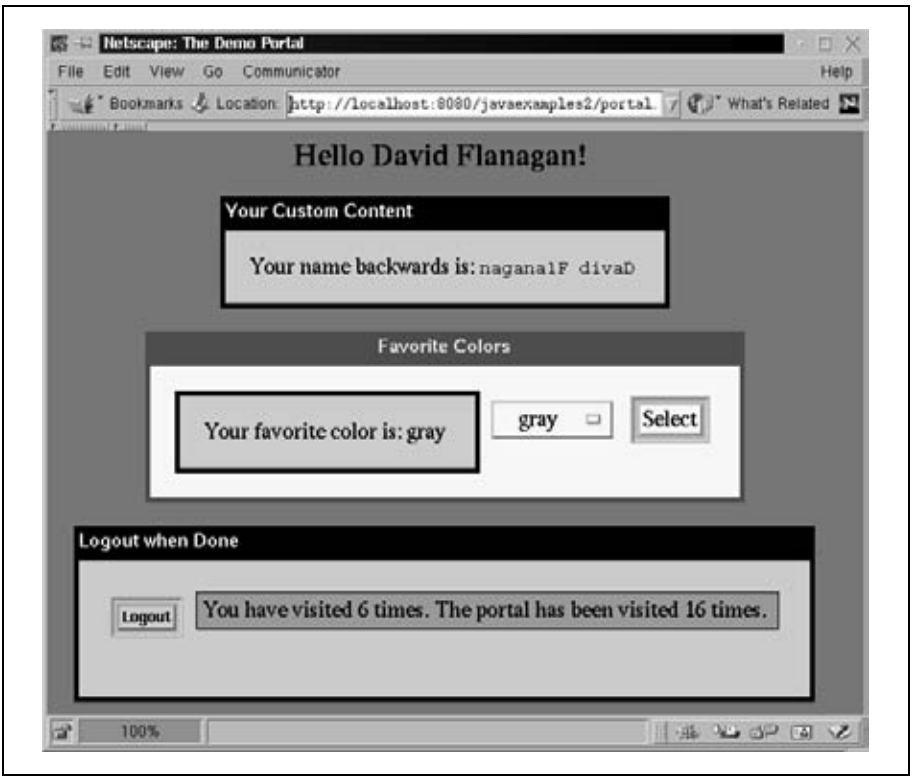


***Рис. 18.4.** Страница portal.jsp*

возможностей (и Java-кода, реализующего их) из JSP-страницы в отдельный класс Java, что еще больше усиливает разделение между кодом Java и HTML. Для создания и манипуляции компонентом в *portal.jsp* используются теги `<jsp:useBean>`, `<jsp:setProperty>` и `<jsp:getProperty>`. Тег `<jsp:useBean>` создает экземпляр компонента и присваивает ему имя. Это имя может использоваться в других тегах JSP и коде Java в выражениях JSP и скриптлетах. Атрибут `scope="session"` указывает, что экземпляр компонента должен быть связан с сеансом пользователя. Другими словами, компонент создается в момент первого посещения страницы пользователем. В это время он запоминается в объекте сеанса, чтобы он мог быть снова восстановлен при последующих посещениях страницы этим пользователем в пределах этого же сеанса.

Создаваемый нами компонент является экземпляром `UserBean`, простого класса, приведенного в листинге примера 18.7. Этот класс представляет отдельного пользователя портала и содержит методы для возвращения информации об этом пользователе, такой как его любимый цвет и произвольный HTML-текст, который должен быть отображен порталом.

Кроме того, *portal.jsp* демонстрирует еще несколько полезных технологий в программировании JSP. Как и страница *login.jsp*, он содержит директиву `<%@taglib>` и активно использует пользовательский тег `<decor:box>`. Он также использует директиву `<@include%>` для статического включения во время компиляции страницы *forcelogin.jsp*. Как мы видели ранее, это вынуждает пользователя зарегистрироваться (войти в систему) прежде, чем может быть показан какой-либо результат. Кроме того, *portal.jsp* использует тег `<jsp:include>` один раз вверху страницы для динамического включения потока вывода сервлета `Hello` и еще дважды в конце страницы для включения вывода из сервлета `Counter`.

Обратите внимание, что код страницы *portal.jsp* динамически генерирует тег `<select>`, предлагающий на выбор список цветов. В этом коде HTML-теги заключены между скриптлетами Java, формирующими начало и конец Java-цикла `for`. HTML-теги выводятся при каждом проходе цикла. Однако поскольку HTML-теги включают JSP-выражения (`<%=...%>`), они будут при каждом проходе развернуты по-разному. Это относительно известный и мощный прием JSP-программирования, хотя такое тесное сочетание Java и HTML не является идеальным.

И наконец, заметим, что *portal.jsp* содержит HTML-форму, отображающую кнопку **logout**. При ее нажатии загружается сервлет `servlet/logout`. Как работает сервлет `Logout`, вы увидите в примере 18.8.

Пример 18.6. portal.jsp

```
<%@page language='java' contentType='text/html'%> <%-- На каждой JSP-странице --%>

<%-- Указываем библиотеку тегов, используемую в этом файле --%>
<%@taglib uri='http://www.davidflanagan.com/tlds/decor_0_1.tld'
          prefix='decor'%>
```

```
<%-- Включаем JSP-код из forcelogin.jsp при компиляции этой страницы.
  Вложенный файл проверяет, зарегистрирован ли пользователь, а если нет,
  то вызывает страницу login.jsp для получения имени пользователя и пароля.
  Это гарантирует, что атрибут username сеанса будет определен. --%>
<%@include file='forcelogin.jsp'%>

<%-- Объявляем новую переменную user. Она является экземпляром UserBean и
  связывается с объектом сеанса. Если это новый сеанс, то создаем компонент и
  устанавливаем его свойство userName в значение атрибута username сеанса. --%>
<jsp:useBean id='user' scope='session'
             class='com.davidflanagan.examples.servlet.UserBean'>
  <jsp:setProperty name='user' property='userName'
                   value='<%=(String)session.getAttribute("username")%>'/>
</jsp:useBean>

<%-- При каждом отображении страницы устанавливаем значение favoriteColor
 компонента user. Из-за того что параметр value не поддерживается,
 устанавливаем значение этого свойства равным значению параметра
 favoriteColor входящего запроса, если параметр с таким именем существует.
 В противном случае не устанавливаем свойство. --%>
<jsp:setProperty name='user' property='favoriteColor'/>

<%-- Начинаем вывод страницы. Отметим использование JSP
 для включения выражений Java в HTML-тег <body>.
 Это явное использование объявленного выше компонента user. --%>
<head><title>The Demo Portal</title></head>
<body bgcolor='<%=user.getFavoriteColor()%>'>

<%-- Приветствуем пользователя путем вызова сервлета Hello и включения сюда
 его вывода. Сервлет Hello рассчитывает найти в объекте сеанса
 имя пользователя, куда оно было записано страницей login.jsp. Обратите
 внимание на разницу между тегом времени исполнения <jsp:include>
 и директивой @include, использованной выше для включения во время
 компиляции файла forcelogin.jsp. --%>
<div align=center>
<h1><jsp:include page='servlet/hello' flush='true'/></h1>
</div>

<%-- Выводим рамку при помощи нашей библиотеки тегов decor, а в качестве
 ее содержимого используем значение свойства customContent компонента user.
 Надеемся, что компонент предоставит интересное содержание, заданное
 указанным пользователем. --%>
<decor:box title='Your Custom Content'->   <%-- Ваш собственный текст --%>
   <jsp:getProperty name='user' property='customContent'/>
</decor:box>

<%-- Начало еще одной рамки --%>
<%-- Содержимым этой рамки будет таблица из двух расположенных рядом ячеек --%>
<p>
<decor:box color='yellow' margin='15'borderColor='darkgreen'
           title='Favorite Colors' titleAlign='center' -titleColor='#aaffaa'>
<table><tr><td>
<%-- Первая ячейка – это еще одна рамка. В ней выводятся некоторый текст
 и значение свойства 'favoriteColor' компонента 'user' --%>
```

```
<decor:box color='pink' margin='20'>
Your favorite color is:
<jsp:getProperty name='user' property='favoriteColor'/>
</decor:box>
</td>
<%-- Это вторая ячейка таблицы. Она содержит форму для выбора любимого цвета.
 Отметим способ, используемый для вывода элемента <select> данной формы. --%>
<td>
<form method='post'>          <%-- Начало формы --%>
<select name='favoriteColor'> <%-- Начало объекта <select> --%>
<%                                // Начало скриптлета Java
// Запрашиваем у компонента список доступных для выбора
// цветов и любимый цвет пользователя.
String[] colors = user.getColorChoices();
String favorite = user.getFavoriteColor();
// Начинаем цикл по списку цветов. Тело этого цикла
// составляет HTML- и JSP-код, расположенный ниже.
for(int i = 0; i < colors.length; i++) {
%>                                <%-- Конец скриптлета --%>
   <%-- При каждом проходе цикла мы будем выводить элемент
   <option>. Отметьте использование тегов выражения JSP
   <%=...%> для вывода оригинального значения тега на каждой итерации цикла. --%>
   <option value='<%=colors[i]%>'
       <%=(colors[i].equals(favorite))?"selected":""%>
    >
       <%=colors[i]%>             <%-- Надпись элемента <option> --%>
   </option>
<%}%>                             <%-- Конец цикла for --%>
</select>                         <%-- Конец объекта <select> --%>
<input type=submit value='Select'> <%-- Кнопка Submit для формы --%>
</form>                           <%-- Конец формы --%>
</td></tr></table>                <%-- Конец второй ячейки и таблицы --%>
</decor:box>                      <%-- конец рамки --%>

<%-- Начало новой рамки. Она также содержит таблицу с двумя ячейками --%>
<br>
<decor:box title='Logout when Done' color='lightblue'>
<table><tr><td valign=top>

<%-- Первая ячейка таблицы – это простая форма, позволяющая
     пользователю выйти --%>
<%-- Дополнительную информацию см. в Logout.java  --%>
<form action='servlet/logout' method='post'>
   <input type=hidden name='page' value='../portal.jsp'>
   <font face='helvetica'><b><input type='submit' value='Logout'></b></font>
</form>

<%-- Вторым элементом рамки является еще одна рамка, отображающая счетчики.
 Мы используем еще два тега jsp:include для вызова сервлета Counter. Обратите
 внимание, что во втором включении мы полагаемся на то, что файл web.xml
 связывает с сервлетом Counter любой URL, заканчивающийся на ".count". --%>
 </td><td valign=top>
```

```
<decor:box color='#a0a0c0' margin='5' borderWidth='1'>
You have visited
   <jsp:include page='servlet/counter' flush='true'>
        <jsp:param name='counter'
               value='<%="portaluser_"+session.getAttribute("username")%>'/>
   </jsp:include>
times. The portal has been visited
   <jsp:include page='portal.jsp.count' flush='true'/>
times.
</decor:box>
</td></tr></table>
</decor:box>
```

В примере 18.7 приведен листинг *UserBean.java* класса JavaBeans, используемого страницей *portal.jsp*. Это стандартный Java-класс с методами доступа `get` и `set`, которые соответствуют соглашениям об именах, принятым в JavaBeans. Обратите внимание, что реализация метода `getCustomContent()` особенно проста: реальное приложение веб-портала должно предоставлять гораздо более значительную информацию, чем данный метод.

Одна из интересных особенностей класса `UserBean` состоит в том, что он реализует интерфейс `HttpSessionBindingListener`. Это означает, что когда экземпляр записывается в объект `HttpSession` сервлета, происходит вызов его метода `valueBound()`, а когда экземпляр удаляется из сеанса, вызывается метод `valueUnbound()`. Например, `UserBean` удаляется из сеанса только при уничтожении самого сеанса либо из-за того, что пользователь завершает работу или становится неактивным, либо из-за выключения самого сервера. В каждом из этих случаев метод `valueUnbound()` позволяет компоненту сохранить свое состояние в каком-нибудь постоянном хранилище, из которого оно впоследствии может быть восстановлено методом `setUserName()`. Методы `valueUnbound()` и `setUserName()` данного класса на текущий момент не выполняют сохранение и восстановление, но в более мощной реализации они могли бы делать это.

Пример 18.7. UserBean.java

```
package com.davidflanagan.examples.servlet;
import javax.servlet.http.*;

/**
 * Данный класс является простым невизуальным компонентом JavaBean,
 * в котором определены свойства и который можно использовать в JSP-странице
 * с помощью тегов <jsp:useBean>, <jsp:setProperty> и <jsp:getProperty>. Эти
 * JSP-теги позволяют вынести за пределы JSP-страницы значительные части кода
 * и поместить их внутрь файлов Java, где их проще читать, редактировать
 * и поддерживать. В этом примере определены только простейшие свойства,
 * но класс мог бы делать гораздо больше.
 *
 * Этот класс создается страницей portal.jsp и связывается
 * с объектом Session. И для того чтобы получать уведомления
```

```
 * о завершении сеанса (выход пользователя или тайм-аут сеанса),
 * он реализует интерфейс HttpSessionBindingListener
 **/
public class UserBean implements HttpSessionBindingListener {
 String username;              // Имя пользователя, которого мы представляем
 String favorite = colors[0]; // Любимый цвет пользователя,
                               // со значением по умолчанию
 static final String[] colors = { "gray", "lightblue", "pink", "yellow" };

 // Это методы get и set для свойства userName. В реальной
 // программе setUserName(), возможно, искал бы информацию
 // о пользователе в какой-нибудь базе данных.
public String getUserName() { return username; }
 public void setUserName(String username) { this.username = username; }

 // Это методы get и set для свойства favoriteColor
 public String getFavoriteColor() { return favorite; }
 public void setFavoriteColor(String favorite) { this.favorite = favorite; }

 // Возвращаем список разрешенных для выбора пользователем цветов
 public String[] getColorChoices() { return colors; }

 // Это get-метод для свойства customContent. В более сложном примере этот
 // метод мог бы посылать запрос базе данных и возвращать пользователю
 // сводки новостей биржевых котировок. Но не здесь.
public String getCustomContent() {
 return "Your name backwards is: <tt>" +
 new StringBuffer(username).reverse() + "</tt>";
 }


 // Этот метод реализует интерфейс HttpSessionBindingListener.
 // Если экземпляр данного класса связать с объектом HttpSession,
 // то этот метод будет вызываться при отсоединении экземпляра, что обычно
 // происходит при аннулировании сеанса вследствие выхода пользователя
 // (logout) или если пользователь слишком долго был неактивен.
 // В реальном примере этот метод, возможно, будет сохранять информацию
 // о пользователе в базе данных или файле.
public void valueUnbound(HttpSessionBindingEvent e) {
 System.out.println(username + " logged out or timed out." +
 " Favorite color: " + favorite);
 }

 // Часть интерфейса HttpSessionBindingListener. Сейчас она
 // для нас неинтересна.
 public void valueBound(HttpSessionBindingEvent e) {}
}
```

Завершение пользовательского сеанса

Пример портала (пример 18.6) требует, чтобы пользователь зарегистрировался (вошел в систему, log in) при первом посещении, и предоставляет кнопку Logout, позволяющую пользователю выйти из сис-

темы. Эти события входа и выхода соответствуют моментам создания и уничтожения объекта `javax.servlet.http.HttpSession`, поддерживающего состояние отдельного пользователя веб-приложения. Протокол HTTP (протокол Интернета) является проколом без состояния: каждый HTTP-запрос не зависим от других, и веб-серверы не поддерживают какое-либо состояние в промежутках между этими запросами. Из-за принципиального отсутствия состояния у базового протокола у контейнеров сервлетов появилась одна полезная особенность – возможность отслеживания сеансов, обычно путем записи невидимого (и безопасного) элемента cookie в клиентский броузер. Элементы cookie содержат уникальный идентификатор, используемый сервлетом для идентификации сеанса.[1]

Сеансы не длятся вечно: если пользователь не присылает запрос веб-приложению в течение заданного в сеансе промежутка времени, контейнер сервлетов завершает этот сеанс. Интервал тайм-аута сеанса является одним из многих настраиваемых параметров, которые могут быть заданы в файле развертывания *WEB-INF/web.xml* (см. пример 18.11).

Кроме того, сеансы могут быть завершены явно, как показано в листинге *Logout.java* из примера 18.8. Это сервлет `Logout`, который вызывается страницей *portal.jsp*, когда пользователь нажимает кнопку **Logout**. Сервлет завершает сеанс вызовом метода `invalidate()` объекта `HttpSession`. Затем он считывает параметр запроса `page` и перенаправляет броузер пользователя на эту страницу. В примере *portal.jsp* параметр `page` был установлен так, чтобы вернуть броузер обратно на *portal.jsp*. Однако поскольку сеанс уже не активен, новый запрос страницы портала является первым запросом нового сеанса, и портал передает его странице *login.jsp*.

Пример 18.8. Logout.java

```
package com.davidflanagan.examples.servlet;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * Этот простой сервлет завершает текущий сеанс и перенаправляет
 * пользовательский броузер по URL, указанному в параметре запроса page.
 * Он может пригодиться для использования во многих веб-приложениях,
 * требующих регистрации пользователей.
 **/
```

[1] Отслеживание сеансов может выполняться даже в случае, когда пользователь запретил использование элементов cookie, при помощи приема, известного как «перезапись URL», в котором уникальные идентификаторы сеансов добавляются в качестве параметров ко всем URL веб-приложения. URL перезаписываются при помощи методов `encodeURL()` и `emcodeRedirectURL()` объекта `HttpServletResponse`.

```
public class Logout extends HttpServlet {
 public void doGet(HttpServletRequest request, HttpServletResponse response)
 throws java.io.IOException
 {
 // Уничтожаем пользовательский сеанс
 request.getSession().invalidate();

 // Выясняем, что отображать дальше
 String nextpage = request.getParameter("page");

 // И перенаправляем броузер пользователя на эту страницу
 response.sendRedirect(nextpage);
 }

 // doPost просто вызывает метод doGet
 public void doPost(HttpServletRequest request,HttpServletResponse response)
 throws java.io.IOException
 {
 doGet(request, response);
 }
}
```

Пользовательские теги

Файлы *login.jsp* и *portal.jsp* используют пользовательский тег `<decor:box>` для рисования цветных прямоугольников с рамкой и необязательным заголовком. Реализация этого тега состоит из двух частей. Первая часть – это файл *DecorBox.java*, показанный в примере 18.9, который содержит Java-код, реализующий этот тег. Второй частью является файл *WEB-INF/tlds/decor_0_1.tld*, показанный в примере 18.10, который представляет собой файл-дескриптор библиотеки тегов (tag library descriptor, TLD), описывающий библиотеку тегов Decor, и содержащиеся в ней теги и их атрибуты. С библиотекой тегов связан также один сложный момент: отображение (mapping) URI, уникально идентифицирующего библиотеку тегов, на локальную копию файла TLD. Это отображение входит в состав файла развертывания веб-приложения *WEB-INF/web.xml*. Мы увидим это в примере 18.11.

Создание пользовательского тега

В программе *DecorBox.java* реализован простой пользовательский тег. При встрече с тегом `<decor:box>` вызывается метод `doStartTag()`, а при встрече с закрывающим тегом `</decor:box>` вызывается метод `doEndTag()`. Для создания требуемого эффекта прямоугольника эти методы используют вложенные HTML-таблицы. Обратите внимание, что для ряда свойств типа `String` в классе определены методы `set`. Эти свойства определяют атрибуты, поддерживаемые тегом. Например, если в теге `<decor:box>` используется атрибут `margin="10"`, перед вызовом `doStartTag()` будет вызван метод `setMargin()` со значением параметра «10».

Пример 18.9. DecorBox.java

```
package com.davidflanagan.examples.servlet;
import javax.servlet.jsp.*;        // Классы JSP
import javax.servlet.jsp.tagext.*; // Классы библиотеки тегов
import java.io.IOException;

/**
 * Эта реализация тега является частью библиотеки тегов "decor".
 * Для отображения декоративной рамки (которая может содержать заголовок)
 * вокруг ее содержимого используются HTML-таблицы. Различные свойства
 * соответствуют атрибутам, поддерживаемым этим тегом.
 **/
public class DecorBox extends TagSupport {
 // Эти поля содержат значения, управляющие внешним видом
 // прямоугольника
 String align;       // Выравнивание прямоугольника
 String title;       // Заголовок прямоугольника
 String titleColor;  // Цвет текста заголовка
 String titleAlign;  // Выравнивание заголовка относительно прямоугольника
 String color;       // Фоновый цвет прямоугольника
 String borderColor; // Фоновый цвет заголовка и цвет рамки
 String margin;      // Расстояние между краем прямоугольника
                     // и его содержимым (пикселы)
 String borderWidth; // Ширина рамки прямоугольника (пикселы)

 // Следующие методы установки свойств устанавливают значения полей,
 // перечисленных выше. Когда страница JSP встретит этот тег, все
 // установки его атрибутов будут преобразованы в вызовы этих методов.
 public void setAlign(String value) { align = value; }
 public void setTitle(String value) { title = value; }
 public void setTitleColor(String value) { titleColor = value; }
 public void setTitleAlign(String value) { titleAlign = value; }
 public void setColor(String value) { this.color = value; }
 public void setBorderColor(String value) { borderColor = value; }
 public void setMargin(String value) { margin = value; }
 public void setBorderWidth(String value) { borderWidth = value; }

 /**
 * Этот унаследованный метод всегда является первым методом установки
 * свойств, вызываемым контейнером JSP. Здесь мы не заботимся о содержимом
 * страницы, а используем этот метод для установки значений по умолчанию
 * для различных свойств. Они инициализируются здесь на тот случай,
 * если контейнер JSP захочет многократно использовать этот тег
 * на нескольких страницах.
 **/
 public void setPageContext(PageContext context) {
 // Важно! Даем возможность базовому классу сохранить объект
 // контекста страницы. Он понадобится нам далее в методе doStartTag().
 super.setPageContext(context);

 // Теперь задаем значения по умолчанию для всех остальных свойств
 align = "center";
```

```
title = null;
titleColor = "white";
titleAlign = "left";
color = "lightblue";
borderColor = "black";
margin = "20";
borderWidth = "4";
}

/**
* Этот метод вызывается, когда встречается тег <decor:box>.
* Перед этим все атрибуты будут обработаны путем вызова
* приведенных выше методов установки свойств.
**/
public int doStartTag() throws JspException {
try {
// Получаем поток вывода из объекта PageContext, который
// был передан через метод setPageContext().
JspWriter out = pageContext.getOut();

// Выводим HTML-теги, необходимые для отображения прямоугольника.
// Тег <div> задает выравнивание, а <table> создает рамку.
out.print("<div align='" + align + "'>" +
"<table bgcolor='" + borderColor + "' " +
"border='0' cellspacing='0' " +
"cellpadding='" + borderWidth + "'>");

// Если есть заголовок, выводим его в качестве ячейки внешней таблицы
if (title != null)
out.print("<tr><td align='" + titleAlign + "'>" +
"<font face='helvetica' size='+1' " +
"color='" + titleColor + "'><b>" +
title + "</b></font></td></tr>");

// Теперь начинаем внутреннюю таблицу, цвет которой
// отличается от цвета рамки.
out.print("<tr><td><table bgcolor='" + color + "' " +
"border='0' cellspacing='0' " +
"cellpadding='" + margin + "'><tr><td>");
}
catch (IOException e) {
// В отличие от PrintWriter, JspWriter может генерировать
// исключение IOException. Нам следует перехватить его
// и "обернуть" исключением JSPException
throw new JspException(e.getMessage());
}

// Это возвращаемое значение предписывает JSP-классу обработать тело тега.
return EVAL_BODY_INCLUDE;
}

/**
* Этот метод вызывается, когда встречается закрывающий тег </decor:box>
**/
```

```
public int doEndTag() throws JspException {
// Пытаемся вывести HTML для закрытия тегов <table> и <div>.
// Перехватываем IOException и повторно генерируем его в виде JspException
try {
JspWriter out = pageContext.getOut();
out.println("</td></tr></table></td></tr></table></div>");
}
catch (IOException e) { throw new JspException(e.getMessage()); }

// Возвращение этого значения ведет к продолжению обработки страницы JSP.
return EVAL_PAGE;
}
}
```

Развертывание пользовательского тега

Перед тем как вы сможете применять пользовательский тег, определенный в *DecorBox.java*, вы должны создать TLD для библиотеки пользовательских тегов. В примере 18.10 показан файл TLD для нашей библиотеки тегов (состоящей из единственного тега для прямоугольника). Как вы видите, TLD представляет собой XML-файл. В нем содержится информация о библиотеке, обо всех входящих в нее тегах и всех атрибутах, поддерживаемых каждым тегом.

Тег `<uri>`, находящийся недалеко от начала TLD, очень важен. Он определяет глобально уникальное имя, под которым становится известной эта библиотека. Имя включает номер версии, для того чтобы эту черновую версию библиотеки можно было отличить от более поздних версий. Целью задания URI является присвоение имени библиотеке тегов. Кроме того, URI может, но не обязан, выполнять функции официального локатора загрузки (download location) для данного файла TLD. Этот URI является именем, которое должно идентифицировать требуемую библиотеку в директиве `<%@taglib>` страницы JSP. Как увидите далее, файл *WEB-INF/web.xml* предоставляет возможность задавать отображения URI библиотек тегов на локальные файлы TLD.[1]

Пример 18.10. WEB-INF/tlds/decor_0_1.tld

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
       PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
```

[1] Существуют и другие способы указания местоположения TLD. Если вы откажетесь от идеи использования для идентификации библиотеки тегов глобально уникальных URI, вы можете указать в директиве `<%@taglib%>` локальный URL, указывающий непосредственно на локальную копию TLD. Или вместо задания положения самого файла TLD вы можете указать положение JAR-файла, содержащего реализацию библиотеки тегов. В этом случае контейнер JSP будет искать файл TLD в каталоге *META-INF/*. За подробной информацией обращайтесь к спецификации JSP.

```
      "http://java.sun.com/j2ee/dtds/web-jsptaglib_1_1.dtd">

<!-- Приведенные выше теги говорят, что это -->
<!-- XML-документ, и формально указывают тип документа -->

<taglib>                        <!-- Определяем библиотеку тегов -->
 <tlibversion>0.1</tlibversion> <!-- Версия этой библиотеки -->
 <jspversion>1.1</jspversion>   <!-- Версия JSP -->
 <shortname>decor</shortname>   <!-- Общее имя библиотеки -->
 <uri>                          <!-- URL, уникально идентифицирующий ее -->
    http://www.davidflanagan.com/tlds/decor_0_1.tld
 </uri>
 <info>                         <!-- Простое описание библиотеки -->
   Простая библиотека тегов для создания декоративного HTML-вывода
 </info>

 <!-- Тег <tag> определяет отдельный тег библиотеки -->
 <tag>
   <!-- Сначала задаем имя тега, класс реализации и описание -->
   <name>box</name>
   <tagclass>com.davidflanagan.examples.servlet.DecorBox</tagclass>
   <info>Отображает цветной прямоугольник с рамкой</info>

   <!-- Затем определяем все поддерживаемые этим тегом атрибуты -->
   <!-- Каждому задаем имя, указываем, является ли он -->
   <!-- обязательным и может ли значение тега быть -->
   <!-- задано в конструкции JSP <%= %> -->
   <attribute>
     <name>align</name>              <!-- Атрибут 'align' -->
     <required>false</required>      <!-- Он необязательный -->
     <rtexprvalue>true</rtexprvalue> <!-- Может иметь значение <%= %> -->
   </attribute>
   <attribute>                       <!-- И т. д., и т. п. -->
     <name>color</name>
     <required>false</required>
     <rtexprvalue>true</rtexprvalue>
   </attribute>
   <attribute>
     <name>borderColor</name>
     <required>false</required>
     <rtexprvalue>true</rtexprvalue>
   </attribute>
   <attribute>
     <name>margin</name>
     <required>false</required>
     <rtexprvalue>true</rtexprvalue>
   </attribute>
   <attribute>
     <name>borderWidth</name>
     <required>false</required>
     <rtexprvalue>true</rtexprvalue>
   </attribute>
   <attribute>
```

```
    <name>title</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>titleColor</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>titleAlign</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
 </tag>
</taglib>
```

Развертывание веб-приложения

В этой главе мы создали несколько сервлетов, несколько JSP-страниц, компонент JavaBeans и библиотеку тегов с TLD. Для того чтобы связать все эти примеры в одно «веб-приложение», вам потребуется создать файл *WEB-INF/web.xml*, описывающий это веб-приложение, и разместить все необходимые файлы в соответствующих местах. Необязательным, но полезным следующим шагом может стать упаковка всех этих файлов в один файл WAR, который может легко распространяться и развертываться.

Конфигурирование веб-приложения с помощью web.xml

Файл *web.xml* показан в примере 18.11. Это XML-файл, в тегах которого содержится различного вида информация о веб-приложении. Версия 2.2 спецификации по сервлетам содержит полную информацию о формате и содержимом этого файла. Данный пример демонстрирует наиболее часто используемые теги, оставляя «за кадром» ряд менее популярных тегов. Наиболее важными тегами файла *web.xml* являются теги `<servlet>`. Тег `<servlet>` задает соответствие между именем сервлета и классом сервлета и определяет для него параметры инициализации (вспомните, что сервлеты обычно считывают свои параметры инициализации в методе `init()`). Для того чтобы заставить веб-приложение работать в вашей системе, вам необходимо изменить значения некоторых параметров инициализации. В частности, следует отредактировать параметр `countfile` сервлета `counter`, а всем параметрам инициализации JDBC сервлета `query` должны быть заданы значения, соответствующие используемому вами серверу базы данных.

Заметим, что совершенно правомерно указывать несколько имен для одного класса реализации сервлета. Каждый тег <servlet> определяет отдельный экземпляр класса сервлета, а у каждого экземпляра может быть свой набор параметров инициализации. Например, если вы захотите использовать сервлет Query для общения с двумя разными серверами баз данных, вы можете использовать два отдельных тега <servlet>, дающих двум экземплярам класса Query имена queryDB1 и queryDB2.

За последним тегом <servlet> в файле *web.xml* идет тег <servlet-mapping>.[1] Этот тег служит для задания соответствия между префиксами или суффиксами URL и отдельными экземплярами сервлетов. Когда веб-сервер принимает запрос на какой-либо URL, удовлетворяющий заданному шаблону, он вызывает связанный с ним сервлет. В своем примере я использовал тег <servlet-mapping>, чтобы сопоставить все URL, оканчивающиеся на .count, сервлету counter.

Тег <session-config> задает информацию, относящуюся к управлению сеансами. В этом примере он указывает, что пользовательский сеанс завершается через 15 минут, в течение которых от пользователя не пришло ни одного запроса. За этим тегом следует тег <taglib>, задающий отображение уникального URI пользовательской библиотеки тегов на локальную копию файла TLD этой библиотеки.

Пример 18.11. web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>   <!-- Это XML-файл -->
<!-- Это тип документа этого XML-файла -->
<!DOCTYPE web-app
   PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
   "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<!-- Описываем конфигурацию и информацию по развертыванию для веб-приложения -->
<web-app>
   <!-- Каждый тег <servlet> определяет информацию о сервлете. -->
   <!-- Этот задает имя для отдельного класса реализации  -->
   <!-- сервлета. Он связывает путь /servlet/hello -->
   <!-- (относительно корня приложения) с указанным классом -->
   <servlet>
     <servlet-name>hello</servlet-name>
     <servlet-class>com.davidflanagan.examples.servlet.Hello</servlet-class>
   </servlet>

   <!-- Это еще один тег <servlet>. Он определяет также -->
   <!-- параметры инициализации, которые читаются сервлетом -->
   <!-- в его методе init(). -->
   <servlet>  <!-- Сервлет counter -->
     <servlet-name>counter</servlet-name>
```

[1] XML DTD для файла *web.xml* требует определенного порядка следования тегов. Вы не можете менять их местами, а должны строго следовать порядку, показанному в этом примере.

```
    <servlet-class>com.davidflanagan.examples.servlet.Counter</servlet-class>
    <init-param>
      <param-name>countfile</param-name>     <!-- Где сохранять состояние -->
      <param-value>/tmp/counts.ser</param-value> <!-- Настройте для вашей
                                                    системы -->
    </init-param>
    <init-param>
      <param-name>saveInterval</param-name>     <!-- Как часто сохранять -->
      <param-value>30000</param-value>          <!-- Каждые 30 секунд -->
    </init-param>
  </servlet>

  <servlet>  <!-- Сервлет query -->
  <servlet-name>query</servlet-name>
  <servlet-class>com.davidflanagan.examples.servlet.Query</servlet-class>
  <!-- Сконфигурируйте все эти параметры для вашей базы данных -->
  <init-param>
    <param-name>driverClassName</param-name>           <!-- Имя класса
                                                          JDBC-драйвера -->
    <param-value>org.gjt.mm.mysql.Driver</param-value> <!-- Драйвер
                                                          для mysql -->
  </init-param>
  <init-param>
    <param-name>url</param-name>          <!-- URL для базы данных -->
    <param-value>jdbc:mysql://dbserver.my.domain.com/dbname</param-value>
  </init-param>
  <init-param>
    <param-name>username</param-name>     <!-- Пользователь базы данных -->
    <param-value>david</param-value>
  </init-param>
  <init-param>
    <param-name>password</param-name>     <!-- Пароль базы данных -->
    <param-value>secret</param-value>
  </init-param>
  </servlet>

<!-- Заметьте, что вы можете определить несколько -->
<!-- именованных экземпляров одного класса сервлета. -->
<!-- Если у вас, например, есть две различные базы -->
<!-- данных, вы могли бы создать еще один экземпляр -->
<!-- сервлета Query с другим именем и другим набором -->
<!-- параметров инициализации -->
<servlet>  <!-- Другой сервлет query -->
  <servlet-name>queryOtherDatabase</servlet-name>
  <servlet-class>com.davidflanagan.examples.servlet.Query</servlet-class>
  <!-- Добавьте сюда параметры инициализации, иначе ваш -->
  <!-- сервлет не будет работать правильно -->
</servlet>

<servlet>  <!-- Сервлет logout -->
  <servlet-name>logout</servlet-name>
  <servlet-class>com.davidflanagan.examples.servlet.Logout</servlet-class>
</servlet>
```

```
  <!-- При связывании сервлетов указываются префиксы и -->
  <!-- суффиксы URL, которые приводят к вызову -->
  <!-- заданного сервлета. Здесь указано, что все URL, -->
  <!-- оканчивающиеся на ".count", будут вызывать -->
  <!-- сервлет "counter"-->
  <servlet-mapping>
    <servlet-name>counter</servlet-name>    <!-- Имя из тега <servlet> -->
    <url-pattern>*.count</url-pattern>      <!-- Какие URL его вызывают -->
  </servlet-mapping>

  <!-- В этом теге указана информация по управлению -->
  <!-- сеансами нашего веб-приложения -->
  <session-config>
    <session-timeout>15</session-timeout>   <!-- "тайм-аут" после 15 минут
                                                 простоя -->
  </session-config>

  <!-- Информация о библиотеках тегов, используемых в приложении -->
  <taglib>
  <!-- Если вы видите этот уникальный идентификатор библиотеки тегов,... -->
    <taglib-uri>http://www.davidflanagan.com/tlds/decor_0_1.tld</taglib-uri>
  <!-- ...используйте эту локальную копию файла TLD -->
    <taglib-location>tlds/decor_0_1.tld</taglib-location>
  </taglib>
</web-app>
```

Упаковка веб-приложений в файлы WAR

В соответствии с версией 2.0 спецификации сервлетов веб-приложения могут распространяться в архивах WAR. Архив WAR – это просто JAR-файл, содержащий информацию по определенному веб-приложению, в особенности файл *WEB-INF/web.xml*. И WAR-, и JAR-архивы используют стандартный ZIP-формат для архивирования и сжатия.

Корневой каталог архива WAR и все его подкаталоги, за исключением каталога *WEB-INF*, содержат видимые для пользователя файлы, которые могут быть показаны для него веб-сервером. Например, вы можете поместить в корневой каталог статический файл *index.html* вашего приложения. Если веб-приложению понадобится много статического содержимого, вы можете распределить его по подкаталогам, таким как каталог *images/*.

Корневой каталог (и его подкаталоги) также является местом, куда помещаются страницы JSP. Веб-сервер не пытается обрабатывать их как статические файлы, а оставляет их JSP-контейнеру. Если для вашего приложения по умолчанию назначен файл *index.jsp*, он должен находиться здесь.

В каталоге */WEB-INF/* хранится конфигурационная информация о веб-приложении и Java-классы, необходимые для работы приложения. Вы уже видели, что */WEB-INF/web.xml* является главным дескриптором развертывания приложения.

Каталог */WEB-INF/classes* содержит все Java-классы, необходимые для веб-приложения. Сюда входят классы сервлетов, компоненты JavaBeans и реализации пользовательских тегов. Обратите внимание, что классы должны находиться в каталогах, соответствующих именам их пакетов. Таким образом, в этом примере на самом деле файлы классов находятся в подкаталоге */WEB-INF/classes/com/davidflanagan/examples/servlet/*.

Каталог */WEB-INF/lib/* содержит все файлы JAR, необходимые веб-приложению. В нем, например, могут находиться пакеты реализации библиотек тегов в формате JAR. Каталог */WEB-INF/tlds/* содержит дескрипторы библиотек тегов, такие как файл */WEB-INF/tlds/decor_0_1.tld*, который описывает библиотеку пользовательских тегов Decor, используемую в JSP-страницах.

Создание файла WAR для веб-приложения в основном сводится к распределению всех файлов по нужным местам и их упаковке. В листинге 18.12 показан сценарий оболочки Unix, который автоматизирует эту задачу. Пользователям Windows не составит труда преобразовать этот сценарий в пакетный файл DOS.

Пример 18.12. makewar.sh: сценарий для упаковки веб-приложения

```
#!/bin/sh

# Удаляем все файлы классов Java и все перекомпилируем
echo "Перекомпилируем классы Java..."
rm *.class
javac *.java

# Создаем временную структуру каталогов для веб-приложения
echo "Создаем WAR-файл..."
mkdir temp
mkdir temp/WEB-INF
mkdir temp/WEB-INF/tlds
mkdir temp/WEB-INF/classes
mkdir temp/WEB-INF/classes/com
mkdir temp/WEB-INF/classes/com/davidflanagan
mkdir temp/WEB-INF/classes/com/davidflanagan/examples
mkdir temp/WEB-INF/classes/com/davidflanagan/examples/servlet

# Теперь копируем наши файлы в эти каталоги
cp *.jsp temp # JSP-файлы помещаются в каталог верхнего уровня
cp WEB-INF/web.xml temp/WEB-INF # Конфигурационные файлы - в каталог WEB-INF/
cp WEB-INF/tlds/decor_0_1.tld temp/WEB-INF/tlds
# Классы Java помещаются в подкаталоги WEB-INF/classes
cp *.class temp/WEB-INF/classes/com/davidflanagan/examples/servlet

# К этому моменту во временном каталоге находятся все наши
# файлы, поэтому мы можем упаковать их в WAR-файл с именем
# javaexamples2.war. Этот файл теперь может быть просто
# скопирован в каталог webapps/ сервера Tomcat.
cd temp
```

```
jar cMf ../javaexamples2.war *

# Теперь удаляем временную структуру каталогов
cd ..
rm -rf temp
```

Упражнения

18-1. Доработайте сервлет `Hello` так, чтобы он никогда не выводил простое приветствие «Hello World». Вместо этого он должен либо приветствовать пользователя по имени, либо спрашивать у него имя, сохранять его в сеансе, а затем приветствовать пользователя, используя это имя. Несмотря на то что вы напишете один сервлет, он должен уметь отображать две разные страницы: страницу приветствия и страницу регистрации (входа в систему). На странице регистрации отображается HTML-форма, и сервер должен уметь обработать данные из этой формы. Использование такой техники показано в классе `DecorBox` для украшения страницы входа в систему.

18-2. Измените сервлет `Hello` снова, как описано в предыдущем упражнении, однако на этот раз не вставляйте HTML-код страниц приветствия и регистрации непосредственно в сервлет. Вместо этого реализуйте содержимое этих страниц в JSP-файлах и используйте сервлет в качестве контроллера, который обрабатывает поток ввода и решает, какая страница должна быть показана. Ваш класс сервлета должен использовать `RequestDispatcher` для передачи запроса соответствующей JSP-странице. Для передачи данных из сервлета JSP-страницам можно использовать объект сеанса или атрибуты запроса. Идею дизайна страницы входа в систему можно позаимствовать из примера *login.jsp*. Однако заметьте, что *login.jsp* отображает страницу и, кроме того, обрабатывает результаты, присланные формой, которую она содержала. В этом упражнении все действия по обработке формы должны производиться непосредственно в сервлете, а страница JSP должна выполнять чисто представительскую функцию и содержать минимальный объем Java-кода.

18-3. Измените сервлет `Counter` так, чтобы он в качестве механизма хранения вместо локального файла использовал базу данных. Напишите сервлет `CounterAdmin`, который будет административным интерфейсом к сервлету `Counter`. Он должен показывать (но не изменять) каждый из счетчиков, находящихся в базе данных. Сервлет `CounterAdmin` должен быть защищен паролем и должен отображать JSP-страницу входа в систему, требующую от пользователя зарегистрироваться перед тем, как будут показаны текущие значения счетчиков. Чтобы задать пароль для сервлета, используйте параметры инициализации из файла *web.xml*.

18-4. Примеры этой главы *login.jsp* и *portal.jsp* демонстрируют ряд способов применения JSP: в них показано, как код Java может быть встроен непосредственно в JSP, а также как в JSP могут использоваться компоненты JavaBeans. Еще одной технологией применения JSP, которая становится достаточно популярной, является архитектура «Модель-Представление-Контроллер» (Model-View-Controller, MVC). В этой архитектуре JavaBeans или другие Java-объекты выполняют функцию хранения данных; они являются «моделью». JSP-страницы обеспечивают «представление» данных и содержат очень мало Java-кода, поэтому они могут создаваться и модифицироваться не программистами на Java, а дизайнерами графики. В состав веб-приложения может входить множество различных JSP-страниц, предоставляющих разнообразное представление данных. Наконец, центральный сервлет играет роль «контроллера», принимая решения, какое представление выводить в ответ на каждый запрос, и используя объект `RequestDispatcher` для передачи этих запросов соответствующей JSP-странице.

В этом упражнении спроектируйте заново пример *portal.jsp* с использованием архитектуры MVC. Напишите отдельный сервлет `Portal`, работающий как контроллер, и создайте, по меньшей мере, три JSP-представления: страницу регистрации (входа в систему), главную страницу портала и страницу, на которой пользователь мог бы выбрать свой любимый цвет. Не забудьте обеспечить возможность выхода пользователя. Используйте (или расширьте) в качестве модели класс `UserBean`, но включите в него механизм сохранения состояния (либо в локальном файле, либо в базе данных). Помните, что страница JSP должна содержать HTML-форму, а в сервлете должны быть реализованы операции по созданию ответа на основе информации, введенной в эту форму. Для веб-приложений на основе форм, таких как наше, полезно включить в каждую HTML-форму скрытое поле, в котором форме или странице назначается уникальное имя, чтобы управляющий сервлет смог определить, с какой формы ему были присланы данные и как их следует обрабатывать.

18-5. Упакуйте сервлет `Portal`, который вы создали в упражнении 18.4, в файл WAR вместе со страницами JSP и всеми остальными нужными ему вспомогательными файлами. Используйте этот WAR-файл для развертывания вашего портала независимо от других примеров этой главы.

Глава 19

XML

XML, или расширяемый язык разметки (Extensible Markup Language), является метаязыком для разметки текстовых документов с помощью структурных тегов, похожих на те, которые содержатся в документах SGML и HTML. XML стал популярным из-за того, что его структурная разметка позволяет документам описывать собственные формат и содержимое. XML вводит понятие «переносимых данных» и может быть достаточно мощным в комбинации с «переносимым кодом», поддерживаемым Java.

Благодаря популярности XML есть множество инструментов, предназначенных для синтаксического анализа и обработки XML-документов. А поскольку XML-документы становятся все более распространенными, вам стоит потратить немного своего времени, чтобы научиться использовать некоторые из этих инструментов при работе с XML. Примеры этой главы познакомят вас с основами анализа и обработки XML. Если вы знакомы с основами структуры XML-файла, у вас не должно возникнуть никаких проблем при их изучении. Обратите внимание, что при работе с XML есть немало тонкостей, и в этой главе не делаются попытки объяснить их все. Для того чтобы больше узнать о XML, обратитесь к книгам O'Reilly: «Java and XML»[1] Бретта Мак-Лахлина (Brett McLaughlin) и «XML Pocket Reference» Роберта Экштейна (Robert Eckstein).

Мир XML и связанных с ним технологий прогрессирует настолько быстро, что может быть сложно даже просто не отстать от сокращений, стандартов, API и номеров версий. В этой главе я попробую дать краткий обзор текущего состояния различных технологий, но предупреждаю,

[1] Бретт Мак-Лахлин «Java и XML», 2-е издание, Символ-Плюс, 2002.

что к тому времени, когда вы будете читать этот материал, некоторые вещи, возможно, уже изменятся, может быть, даже радикально.

Анализ с помощью JAXP и SAX 1

Работа с XML-документом начинается с его синтаксического анализа. Есть два широко используемых подхода к синтаксическому анализу XML: SAX и DOM. Начнем мы с синтаксического анализа SAX, а затем рассмотрим синтаксический анализ DOM. В самом конце главы мы познакомимся с новым, но очень перспективным XML API, основанным на Java и известным как JDOM.

SAX, или Simple API for XML, – это не синтаксический анализатор, а скорее программный интерфейс Java, который описывает работу анализатора. При анализе XML-документа, использующем SAX API, вы создаете класс, который реализует различные методы обработки «событий». По мере того как анализатор встречает различные типы элементов XML-документа, он вызывает соответствующие методы обработчика событий, которые вы определили. Ваши методы предпринимают какие-либо действия, необходимые для выполнения требуемой задачи. В модели SAX синтаксический анализатор преобразует XML-документ в последовательность вызовов методов Java. Этот анализатор не формирует какое-либо дерево (хотя ваши методы могут делать это, если вам это нужно). SAX-анализ обычно довольно эффективен и поэтому является для вас наилучшим выбором при решении простейших задач по обработке XML.

SAX API был создан Дэвидом Меггинсоном (David Megginson), *http://www.megginson.com/SAX/*. Java-реализация этого интерфейса находится в пакете `org.xml.sax` и его подпакетах. SAX стал стандартом дефакто, но так и не был стандартизирован никакой официальной организацией. SAX версии 1 находился в использовании некоторое время, SAX 2 был завершен в мае 2000 г. Между SAX 1 и SAX 2 API есть ряд различий. Существует множество базирующихся на Java XML-анализаторов, совместимых с интерфейсами SAX 1 и SAX 2.

При использовании SAX API вы не можете полностью абстрагироваться от деталей реализации применяемого вами XML-анализатора: как минимум, ваш код должен содержать имя класса используемого анализатора. Это то место, где на сцену выходит JAXP. JAXP означает «Java API for XML Parsing». Это «дополнительный пакет», созданный фирмой Sun, который состоит из пакета `javax.xml.parsers`. JAXP представляет собой тонкий слой, построенный на основе SAX (и на основе DOM, как мы это увидим ниже), который стандартизирует API для получения и использования объектов анализатора SAX (и DOM). Пакет JAXP распространяется с реализациями анализатора по умолчанию, но позволяет с легкостью подключать и конфигурировать при помощи системных свойств и другие анализаторы. Когда писалась эта книга,

текущей версией JAXP была 1.0.1. Она поддерживает SAX 1, но не поддерживает SAX 2. Однако к тому времени, когда вы будете ее читать, возможно, будет доступен JAXP версии 1.1, в который будет включена поддержка для SAX 2.

В примере 19.1 приведен листинг *ListServlets1.java* программы, которая использует JAXP и SAX для синтаксического разбора дескриптора развертывания веб-приложения и выводит список имен сервлетов, сконфигурированных этим файлом. Если вы еще не читали главу 18, вам следует знать, что основанные на сервлетах веб-приложения конфигурируются с помощью XML-файла с именем *web.xml*. Этот файл содержит теги `<servlet>`, которые задают соответствие между именами сервлетов и реализующими их классами Java. Для того чтобы помочь вам понять задачу, решаемую программой *ListServlets1.java*, здесь приведен фрагмент файла *web.xml*, разработанного в главе 18:

```
<servlet>
    <servlet-name>hello</servlet-name>
    <servlet-class>com.davidflanagan.examples.servlet.Hello</servlet-class>
  </servlet>

  <servlet>
    <servlet-name>counter</servlet-name>
    <servlet-class>com.davidflanagan.examples.servlet.Counter</servlet-class>
    <init-param>
      <!-- где сохранять состояние -->
      <param-name>countfile</param-name>
      <!-- настройте под вашу систему -->
      <param-value>/tmp/counts.ser</param-value>
    </init-param>
    <init-param>
      <!-- как часто сохранять -->
      <param-name>saveInterval</param-name>
      <!-- каждые 30 секунд -->
      <param-value>30000</param-value>
    </init-param>
  </servlet>

  <servlet>
    <servlet-name>logout</servlet-name>
    <servlet-class>com.davidflanagan.examples.servlet.Logout</servlet-class>
  </servlet>
```

ListServlets1.java содержит метод `main()`, который использует JAXP API для получения экземпляра SAX-анализатора. Затем он сообщает анализатору, что нужно анализировать, и начинает его выполнение. Остальные методы этого класса вызываются синтаксическим анализатором. Обратите внимание, что `ListServlets1` расширяет SAX-класс `HandlerBase`. Этот суперкласс предоставляет фиктивные реализации для всех методов обработчиков событий SAX. Этот пример просто замещает обработчики, представляющие для него интерес. Синтаксичес-

кий анализатор вызывает метод `startElement()`, когда он считывает XML-тег, и вызывает `endElement()`, когда находит закрывающий тег. Метод `characters()` вызывается тогда, когда синтаксический анализатор считывает строку текста, не содержащую символов разметки. Наконец, при возникновении каких-либо ошибок в процессе синтаксического разбора анализатор вызывает методы `warning()`, `error()` и `fatalError()`. Реализации этих методов написаны специально для извлечения нужной информации из файла *web.xml* и основаны на знании структуры этого типа файла.

Обратите внимание, что файлы *web.xml* несколько необычны тем, что они не используют атрибуты ни в одном из XML-тегов. То есть имена сервлетов задаются тегом `<servlet-name>`, вложенным в тег `<servlet>`, вместо того чтобы просто использовать атрибут `name` в самом теге `<servlet>`. Этот факт делает пример программы несколько более сложным, чем он мог бы быть в другом случае. Файл *web.xml* разрешает использовать атрибуты `id` для всех своих тегов. Хотя контейнеры сервлетов не предполагают использование этих атрибутов, последние могут быть полезны для конфигурационных средств, анализирующих и автоматически генерирующих файлы *web.xml*. Для полноты метод `startElement()` в примере 19.1 ищет в теге `<servlet>` атрибут `id`. Значение этого атрибута, если он существует, сообщается в выводе программы.

Пример 19.1. ListServlets1.java

```
package com.davidflanagan.examples.xml;
import javax.xml.parsers.*;     // Пакет JAXP
import org.xml.sax.*;           // Главный пакет SAX
import java.io.*;

/**
 * Анализ файла web.xml с помощью JAXP и SAX1. Распечатываем
 * имена классов всех сервлетов, перечисленных в этом файле.
 *
 * Этот класс реализует вспомогательный класс HandlerBase.
 * Это означает, что в нем определены все методы обратного
 * вызова ("callback"), которые анализатор SAX будет вызывать для
 * уведомления приложения. В этом примере мы замещаем нужные нам методы.
 *
 * На всем протяжении этого примера используются полные
 * имена пакетов для того, чтобы отличать интерфейсы JAXP и SAX.
 **/
public class ListServlets1 extends org.xml.sax.HandlerBase {
  /** Метод main выполняет подготовку к анализу */
  public static void main(String[] args)
    throws IOException, SAXException, ParserConfigurationException
  {
    // Создаем "фабрику анализаторов" JAXP для создания SAX-анализаторов
    javax.xml.parsers.SAXParserFactory spf=SAXParserFactory.newInstance();

    // Настраиваем фабрику анализаторов на необходимый нам тип анализатора
```

```
    spf.setValidating(false); // Проверка правильности не требуется

    // Используем фабрику анализаторов для создания объекта SAXParser.
    // Заметьте, что SAXParser - это JAXP-класс, а не SAX-класс
    javax.xml.parsers.SAXParser sp = spf.newSAXParser();

    // Создаем входной источник SAX для параметра file
    org.xml.sax.InputSource input=new InputSource(new FileReader(args[0]));

    // Передаем InputSource абсолютный URL этого файла, для того чтобы он
    // смог разрешить относительные URL в объявлении <!DOCTYPE> и т. п.
    input.setSystemId("file://" + new File(args[0]).getAbsolutePath());

    // Создаем экземпляр этого класса; в нем определены все
    // методы-обработчики
    ListServlets1 handler = new ListServlets1();

    // Наконец, указываем анализатору разобрать
    // поток ввода с отправкой уведомлений обработчику
    sp.parse(input, handler);

    // Вместо использования метода SAXParser.parse(), являющегося
    // частью JAXP API, мы могли бы использовать SAX1 API напрямую.
    // Отметим разницу между JAXP-классом javax.xml.parsers.SAXParser
    // и SAX1-классом org.xml.sax.Parser
    //
    // org.xml.sax.Parser parser = sp.getParser(); // Получаем SAX-анализатор
    // parser.setDocumentHandler(handler); // Задаем главный обработчик
    // parser.setErrorHandler(handler);    // Задаем обработчик ошибок
    // parser.parse(input);                // Анализируем!
  }

  // Накапливаем проанализированный текст
  StringBuffer accumulator = new StringBuffer();
  String servletName;    // Имя сервлета
  String servletClass;   // Имя класса сервлета
  String servletId;      // Значение атрибута id тега <servlet>

  // Когда анализатор встречает простой текст (не XML-элемент),
  // он вызывает этот метод, который накапливает текст в буфере строк
  public void characters(char[] buffer, int start, int length) {
    accumulator.append(buffer, start, length);
  }

  // Каждый раз, когда анализатор встречает начало нового элемента,
  // он вызывает этот метод, который очищает буфер строк
  public void startElement(String name, AttributeList attributes) {
    accumulator.setLength(0); // Готовы к накапливанию нового текста
    // Если это тег <servlet>, ищем атрибут id
    if (name.equals("servlet"))
      servletId = attributes.getValue("id");
  }

  // Когда анализатор доходит до конца элемента, он вызывает этот метод
  public void endElement(String name) {
```

```
      if (name.equals("servlet-name")) {
        // После </servlet-name> мы знаем, что сохранено имя сервлета
        servletName = accumulator.toString().trim();
      }
      else if (name.equals("servlet-class")) {
        // После </servlet-class> у нас есть имя класса
        servletClass = accumulator.toString().trim();
      }
      else if (name.equals("servlet")) {
        // Допуская, что документ правильный, после того как мы достигли
        // тега </servlet>, мы знаем, что у нас есть имя сервлета
        // и имя класса, которые можно распечатать
        System.out.println("Servlet " + servletName +
                   ((servletId != null)?" (id="+servletId+")":"") +
                   ": " + servletClass);
      }
    }

    /** Этот метод вызывается при возникновении предупреждений */
    public void warning(SAXParseException exception) {
      System.err.println("WARNING: строка " + exception.getLineNumber()
          + ": " + exception.getMessage());
    }

    /** Этот метод вызывается при возникновении ошибок */
    public void error(SAXParseException exception) {
      System.err.println("ERROR: строка " + exception.getLineNumber()
          + ": " + exception.getMessage());
    }

    /** Этот метод вызывается при возникновении неисправимых ошибок. */
    public void fatalError(SAXParseException exception) throws SAXException {
      System.err.println("FATAL: строка " + exception.getLineNumber()
          + ": " + exception.getMessage());
      throw(exception);
    }
  }
```

Компиляция и выполнение примера

Для того чтобы выполнить предыдущий пример, вам понадобится пакет JAXP от фирмы Sun. Вы можете загрузить его с *http://java.sun.com/xml/*. После загрузки разархивируйте и установите пакет в каком-нибудь подходящем месте в вашей системе. В JAXP версии 1.0.1 в пакете загрузки находятся два файла JAR: *jaxp.jar* (классы JAXP API) и *parser.jar* (SAX и DOM API и реализации анализатора по умолчанию). Чтобы откомпилировать и выполнить этот пример, вам нужно указать в переменной окружения `CLASSPATH` оба этих JAR-файла. Если в вашей переменной `CLASSPATH` указаны какие-либо другие XML-анализаторы, такие как Xerces, удалите их или убедитесь в том, что файлы JAXP указаны первыми; в противном случае вы можете столкнуться с

проблемами конфликта версий разных анализаторов. Отметим, что вы, вероятно, не захотите постоянно менять значение CLASSPATH, так как для следующего примера вам понадобится изменить его снова. Одно простое решение, возможное в Java версии 1.2 и выше, состоит во временном помещении копий JAR-файлов JAXP в каталог *jre/lib/ext/* вашей инсталляции Java.

С двумя JAR-файлами JAXP, временно находящимися в вашей CLASSPATH, вы можете откомпилировать и выполнить *ListServlets1.java* как обычно. При запуске укажите в командной строке имя файла *web.xml*. Вы можете использовать файл, включенный в состав загружаемых примеров для этой книги, или указать файл из вашей собственной машины сервлетов.

В этом примере есть одна сложность. Большинство файлов *web.xml* содержат тег <!DOCTYPE>, который определяет тип документа (или DTD). Несмотря на тот факт, что в примере 19.1 указано, что синтаксический анализатор не должен проверять правильность документа, совместимый XML-анализатор все же должен считывать DTD для всех документов, имеющих объявление <!DOCTYPE>. Большинство *web.xml* содержат объявление, подобное следующему:

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

Для того чтобы считать DTD, анализатор должен уметь читать указанный URL. Если ваша система во время выполнения примера не подключена к Интернету, она может «зависнуть». Один из выходов – заменить URL для DTD именем локальной копии DTD, что было сделано в файле *web.xml*, включенном в состав загружаемых примеров. Другое решение этой проблемы состоит в простом удалении (или закомментировании) объявления <!DOCTYPE> из файла *web.xml*, который вы обрабатываете программой ListServlets1.

Анализ с помощью SAX 2

В примере 19.1 было показано, как вы можете анализировать XML-документ при помощи интерфейса SAX 1, который поддерживается текущей версией JAXP (на момент написания книги). Однако интерфейс SAX 1 является устаревшим, поэтому в примере 19.2 показано, как вы можете выполнять похожую задачу синтаксического анализа при помощи интерфейса SAX 2 и свободно распространяемого анализатора Xerces, доступного на сайте Apache Software Foundation.

В примере 19.2 приведен листинг программы *ListServlets2.java*. Как и пример *ListServlets1.java*, эта программа читает указанный файл *web.xml* и ищет в нем теги <servlet>, так что она может распечатать соответствие имен сервлетов классам сервлетов. Однако этот пример идет

немного дальше, чем предыдущий, и ищет также теги <servlet-mapping>, поэтому он может выводить шаблоны URL, связанные с указанными сервлетами. В примере используются две хеш-таблицы, предназначенные для хранения информации по мере ее накопления, а затем, по завершении анализа, для распечатки всей этой информации.

Работа интерфейса SAX 2 похожа на работу SAX 1, но несколько классов и интерфейсов имеют другие названия, а некоторые методы имеют другие сигнатуры. Многие из этих изменений потребовались для добавления в SAX 2 поддержки пространств имен XML. По мере изучения примера 19.2 обратите внимание на отличия этого интерфейса от интерфейса примера 19.1.

Пример 19.2. ListServlets2.java

```
package com.davidflanagan.examples.xml;
import org.xml.sax.*;          // Главный пакет SAX
import org.xml.sax.helpers.*; // Вспомогательные классы SAX
import java.io.*;              // Для чтения входного файла
import java.util.*;            // Хеш-таблица, списки и т. д.

/**
 * Анализ файла web.xml при помощи SAX2 API и анализатора
 * Xerces из проекта Apache.
 *
 * Этот класс расширяет DefaultHandler для того, чтобы его экземпляры
 * могли служить в качестве обработчиков событий SAX 2 и могли получать
 * уведомления от анализатора о событиях, возникающих в процессе
 * синтаксического анализа. Мы просто замещаем методы, получающие
 * события, которые нас интересуют
 **/
public class ListServlets2 extends org.xml.sax.helpers.DefaultHandler {
  /** Главный метод выполняет подготовку к анализу */
  public static void main(String[] args) throws IOException, SAXException {
    // Создаем анализатор, который мы будем использовать.
    // Реализация анализатора – это класс Xerces, но мы
    // используем его только через SAX XMLReader API
    org.xml.sax.XMLReader parser=new org.apache.xerces.parsers.SAXParser();

    // Указываем, что нам не требуется проверка правильности.
    // Это SAX2 API для настройки поведения анализатора.
    // Отметьте использование в качестве имени свойства глобально
    // уникального URL. В действительности проверка правильности по умолчанию
    // отключена, поэтому эта строка на самом деле необязательна.
    parser.setFeature("http://xml.org/sax/features/validation", false);

    // Создаем экземпляр класса для предоставления обработчиков
    // для анализатора и сообщаем ему об этих обработчиках
    ListServlets2 handler = new ListServlets2();
    parser.setContentHandler(handler);
    parser.setErrorHandler(handler);

    // Создаем входной источник, описывающий файл, подлежащий анализу.
    // Затем указываем анализатору выполнить анализ входного потока
    // из этого источника
```

```
    org.xml.sax.InputSource input=new InputSource(new FileReader(args[0]));
    parser.parse(input);
  }

  // Соответствие имен сервлетов и имен классов сервлетов
  HashMap nameToClass;
  // Соответствие имен сервлетов и шаблонов URL
  HashMap nameToPatterns;

  StringBuffer accumulator;    // Аккумулируем текст
  // Запоминаем текст
  String servletName, servletClass, servletPattern;

  // Вызывается в начале синтаксического разбора.
  // Мы используем его как метод init()
  public void startDocument() {
    accumulator = new StringBuffer();
    nameToClass = new HashMap();
    nameToPatterns = new HashMap();
  }

  // Когда анализатор встречает простой текст (не XML-элементы),
  // он вызывает этот метод, который накапливает текст в буфере строк.
  // Заметьте, что этот метод может быть вызван несколько раз даже
  // при отсутствии промежуточных элементов.
  public void characters(char[] buffer, int start, int length) {
    accumulator.append(buffer, start, length);
  }

  // В начале каждого нового элемента удаляем накопленный текст.
  public void startElement(String namespaceURL, String localName,
                String qname, Attributes attributes) {
    accumulator.setLength(0);
  }

  // Выполняем особые действия, когда мы достигли конца
  // выбранного элемента. Хотя мы не используем анализатор
  // с проверкой правильности, этот метод предполагает,
  // что анализируемый нами файл web.xml правильный.
  public void endElement(String namespaceURL, String localName, String qname)
  {
    // Запоминаем имя сервлета
    if (localName.equals("servlet-name")) {
      servletName = accumulator.toString().trim();
    }
    // Запоминаем класс сервлета
    else if (localName.equals("servlet-class")) {
      servletClass = accumulator.toString().trim();
    }
    // Запоминаем шаблон сервлета
    else if (localName.equals("url-pattern")) {
      servletPattern = accumulator.toString().trim();
    }
    // Сохраняем пару имя/класс
    else if (localName.equals("servlet")) {
```

```
        nameToClass.put(servletName, servletClass);
      }
      // Сохраняем пару имя/шаблон
      else if (localName.equals("servlet-mapping")) {
        List patterns = (List)nameToPatterns.get(servletName);
        if (patterns == null) {
          patterns = new ArrayList();
          nameToPatterns.put(servletName, patterns);
        }
        patterns.add(servletPattern);
      }
    }

    // Вызывается по окончании анализа. Здесь используется
    // для распечатки результатов.
    public void endDocument() {
      List servletNames = new ArrayList(nameToClass.keySet());
      Collections.sort(servletNames);
      for(Iterator iterator = servletNames.iterator(); iterator.hasNext();) {
        String name = (String)iterator.next();
        String classname = (String)nameToClass.get(name);
        List patterns = (List)nameToPatterns.get(name);
        System.out.println("Сервлет: " + name);
        System.out.println("Класс: " + classname);
        if (patterns != null) {
          System.out.println("Шаблоны:");
          for(Iterator i = patterns.iterator(); i.hasNext(); ) {
            System.out.println("\t" + i.next());
          }
        }
        System.out.println();
      }
    }

    // Выводим предупреждение
    public void warning(SAXParseException exception) {
      System.err.println("WARNING: строка " + exception.getLineNumber()
          + ": " + exception.getMessage());
    }

    // Отчет об ошибках анализа
    public void error(SAXParseException exception) {
      System.err.println("ERROR: строка " + exception.getLineNumber()
         + ": " + exception.getMessage());
    }

    // Отчет о неисправимых ошибках и выход
    public void fatalError(SAXParseException exception) throws SAXException {
      System.err.println("FATAL: строка " + exception.getLineNumber()
        + ": " + exception.getMessage());
      throw(exception);
    }
  }
```

Компиляция и выполнение примера

В примере `ListServlets2` используется синтаксический анализатор Xerces-J из Apache XML Project. Этот свободно распространяемый анализатор можно загрузить с *http://xml.apache.org/*. После загрузки распакуйте дистрибутив Xerces-J в подходящем месте в вашей системе. В дистрибутиве вы должны найти файл *xerces.jar*. Для того чтобы можно было откомпилировать и выполнить пример *ListServlets2.java*, этот файл необходимо указать в переменной окружения `CLASSPATH`. Обратите внимание, что файлы *xerces.jar* и *parsers.jar* из дистрибутива JAXP содержат версии классов для DOM и SAX. Вы не должны допускать одновременного размещения этих файлов в вашей `CLASSPATH`.

Анализ и обработка с помощью JAXP и DOM

Первые два примера этой главы для синтаксического анализа XML-документов использовали SAX API. Теперь мы обратимся к другому широко используемому программному интерфейсу синтаксического анализа – DOM, или объектной модели документа (Document Object Model). DOM API является стандартом, опубликованным Консорциумом W3C (World Wide Web Consortium); его Java-реализация состоит из пакета `org.w3c.dom` и его подпакетов. Текущая версия стандарта DOM – Level 1. На момент написания этих строк DOM Level 2 API находился в процессе стандартизирования в W3C.

Объектная модель документа определяет интерфейс для построения дерева из XML-документов. Основные свойства узла этого дерева определены в интерфейсе `org.xml.dom.Node`. Подынтерфейсы, такие как `Document`, `Element`, `Entity` и `Comment`, определяют свойства конкретных типов узлов. Программа, использующая модель анализа DOM, сильно отличается от тех, которые используют SAX. В случае с DOM в вашем распоряжении имеется анализатор, читающий ваш XML-документ и преобразующий его в дерево объектов `Node`. По окончании синтаксического анализа вы можете обходить это дерево в поисках нужной вам информации. Модель анализа DOM бывает полезна, если вам требуется выполнять многократные обходы этого дерева, если вы хотите изменить структуру дерева или если вам необходим произвольный доступ к XML-документу, а не последовательный, предоставляемый моделью SAX.

В примере 19.3 приведен листинг программы *WebAppConfig.java*. Как и первые два примера этой главы, `WebAppConfig` считывает дескриптор развертывания веб-приложения *web.xml*. Этот пример использует синтаксический анализатор DOM для формирования дерева и последующего выполнения над ним некоторых действий для демонстрации того, как вы можете работать с деревом узлов DOM.

Конструктор `WebAppConfig()` использует JAXP API для получения DOM-анализатора, а затем использует этот анализатор для формирования дерева, представляющего XML-файл. Корневой узел этого дерева имеет тип `Document`. Данный объект `Document` хранится в поле экземпляра `WebAppConfig`, поэтому он доступен для навигации и изменения другими методами класса. В состав этого класса также входит метод `main()`, который вызывает все другие методы.

Метод `getServletClass()` выполняет поиск тегов `<servlet-name>` и возвращает текст, находящийся в связанном с ним теге `<servlet-class>`. (В файле *web.xml* эти теги всегда образуют пары.) Этот метод демонстрирует несколько особенностей DOM-дерева, в частности метод `getElementsByTagName()`. Метод `AddServlet()` вставляет в дерево новый тег `<servlet>`. Он демонстрирует, как можно создавать новые DOM-узлы и добавлять их в существующее дерево. Наконец, метод `output()` использует `XMLDocumentWriter` для осуществления навигации по всем узлам дерева и преобразования их обратно в формат XML. Класс `XMLDocumentWriter` рассматривается в следующем разделе и приведен в листинге 19.3.

Пример 19.3. WebAppConfig.java

```
package com.davidflanagan.examples.xml;
import javax.xml.parsers.*;  // Классы JAXP для анализа
import org.w3c.dom.*;        // Классы W3C DOM для навигации по документу
import org.xml.sax.*;        // Классы SAX, используемые
                             // для обработки ошибок с помощью JAXP
import java.io.*;            // Для чтения входного файла

/**
 * Объект WebAppConfig является "оберткой" дерева DOM
 * для файла web.xml. Методы этого класса используют DOM
 * API для работы с деревом различными способами.
 **/
public class WebAppConfig {
  /** Метод main() создает и демонстрирует объект WebAppConfig */
  public static void main(String[] args)
    throws IOException, SAXException, ParserConfigurationException
  {
    // Создаем новый объект WebAppConfig, который будет представлять
    // файл web.xml, переданный в качестве первого аргумента командной строки
    WebAppConfig config = new WebAppConfig(new File(args[0]));
    // Запрашиваем у дерева имя класса, связанного с
    // заданным именем сервлета
    System.out.println("Класс для сервлета " + args[1] + " - " +
                       config.getServletClass(args[1]));
    // Вводим в дерево DOM новое соответствие имя/класс для сервлета
    config.addServlet("foo", "bar");
    // И записываем XML-версию дерева DOM в стандартный выходной поток
    config.output(new PrintWriter(new OutputStreamWriter(System.out)));
  }

  // Это поле содержит разобранное (проанализированное) дерево DOM
```

```
org.w3c.dom.Document document;

/**
 * В этот конструктор передается XML-файл. Он использует JAXP API
 * для получения анализатора DOM и для преобразования этого файла в объект
 * документа DOM, который используется во всех остальных методах класса.
 **/
public WebAppConfig(File configfile)
  throws IOException, SAXException, ParserConfigurationException
{
  // Получаем объект-фабрику анализаторов JAXP
  javax.xml.parsers.DocumentBuilderFactory dbf =
    DocumentBuilderFactory.newInstance();
  // Сообщаем фабрике, анализатор какого типа нам нужен
  dbf.setValidating(false);
  // Используем фабрику для получения объекта-анализатора JAXP
  javax.xml.parsers.DocumentBuilder parser = dbf.newDocumentBuilder();

  // Сообщаем анализатору, как следует обрабатывать ошибки.
  // Заметим, что в JAXP API анализаторы DOM используют
  // для обработки ошибок SAX API
  parser.setErrorHandler(new org.xml.sax.ErrorHandler() {
      public void warning(SAXParseException e) {
        System.err.println("WARNING: " + e.getMessage());
      }
      public void error(SAXParseException e) {
        System.err.println("ERROR: " + e.getMessage());
      }
      public void fatalError(SAXParseException e)
        throws SAXException {
        System.err.println("FATAL: " + e.getMessage());
        throw e;  // Повторно генерируем исключение
      }
    });

  // Наконец, используем JAXP-анализатор для разбора файла.
  // Этот вызов возвращает объект Document. Теперь, когда
  // у нас есть этот объект, оставшаяся часть класса использует
  // для работы с ним DOM API; JAXP больше не требуется.
  document = parser.parse(configfile);
}

/**
 * Этот метод ищет в дереве DOM заданные узлы Element, чтобы выяснить
 * имя класса, ассоциированное с указанным именем сервлета
 **/
public String getServletClass(String servletName) {
  // Находим все элементы <servlet> и выполняем по ним цикл
  NodeList servletnodes = document.getElementsByTagName("servlet");
  int numservlets = servletnodes.getLength();
  for(int i = 0; i < numservlets; i++) {
    Element servletTag = (Element)servletnodes.item(i);
    // Получаем первый тег <servlet-name>, находящийся внутри тега <servlet>
```

```
    Element nameTag = (Element)
      servletTag.getElementsByTagName("servlet-name").item(0);
    if (nameTag == null) continue;

    // Тег <servlet-name> должен иметь единственный дочерний элемент
    // типа Text. Получаем этот элемент и извлекаем из него текст.
    // Используем функцию trim() для отсечения пробелов в начале и конце.
    String name =((Text)nameTag.getFirstChild()).getData().trim();

    // Если этот тег <servlet-name> имеет правильное имя,
    if (servletName.equals(name)) {
      // Получаем соответствующий тег <servlet-class>
      Element classTag = (Element)
        servletTag.getElementsByTagName("servlet-class").item(0);
      if (classTag != null) {
        // Извлекаем текст тега и возвращаем его
        Text classTagContent = (Text)classTag.getFirstChild();
        return classTagContent.getNodeValue().trim();
      }
    }
  }

  // Если мы здесь, значит, соответствующих имен сервлетов нет
  return null;
}

/**
 * Этот метод добавляет в документ новое соответствие
 * имя/класс в форме поддерева <servlet>
 **/
public void addServlet(String servletName, String className) {
  // Создаем тег <servlet>
  Element newNode = document.createElement("servlet");
  // Создаем теги <servlet-name> и <servlet-class>
  Element nameNode = document.createElement("servlet-name");
  Element classNode = document.createElement("servlet-class");
  // Добавляем к этим тегам текстовые элементы "имя" и "имя класса"
  nameNode.appendChild(document.createTextNode(servletName));
  classNode.appendChild(document.createTextNode(className));
  // И добавляем эти теги к тегу <servlet>
  newNode.appendChild(nameNode);
  newNode.appendChild(classNode);

  // Сейчас, когда у нас есть новое поддерево, выясняем, куда его
  // поместить. Этот код ищет еще один тег <servlet> и вставляет
  // новый непосредственно перед ним. Заметим, что этот код завершится
  // аварийно, если документ еще не содержит ни одного тега <servlet>.
  NodeList servletnodes = document.getElementsByTagName("servlet");
  Element firstServlet = (Element)servletnodes.item(0);

  // Вставляем новый узел перед первым узлом <servlet>
  firstServlet.getParentNode().insertBefore(newNode, firstServlet);
}
```

```
    /**
     * Выводим дерево DOM в указанный поток как XML-документ.
     * Подробности см. в примере XMLDocumentWriter.
     **/
    public void output(PrintWriter out) {
      XMLDocumentWriter docwriter = new XMLDocumentWriter(out);
      docwriter.write(document);
      docwriter.close();
    }
}
```

Компиляция и выполнение примера

Класс `WebAppConfig` использует интерфейсы JAXP и DOM, поэтому у вас в переменной окружения `CLASSPATH` должны быть указаны оба файла, *jaxp.jar* и *parser.jar*, из дистрибутива JAXP. Вам следует избегать одновременного указания там JAR-файла Xerces, иначе вы можете столкнуться с проблемами несоответствия версии между анализатором DOM Level 1 JAXP 1.0 и анализатором DOM Level 2 Xerces. Откомпилируйте *WebAppConfig.java* как обычно. Для того чтобы выполнить программу, задайте в качестве первого параметра командной строки имя файла *web.xml* для анализа, а в качестве второго укажите имя сервлета. Когда вы запустите программу, она напечатает имя класса (если он существует), который соответствует указанному имени сервлета. Затем она вставит в дерево фиктивный тег `<servlet>` и выведет на стандартный вывод измененное дерево в формате XML. Вам, вероятно, потребуется перенаправить выходной поток программы утилите страничного разбиения, такой как *more*.

Навигация по дереву DOM

Класс `WebAppConfig` из примера 19.3 преобразует XML-файл в дерево DOM, изменяет его, а затем конвертирует обратно в XML-файл. Он делает это при помощи класса `XMLDocumentWriter`, который приведен в примере 19.4. Метод `write()` этого класса рекурсивно проходит дерево DOM от узла к узлу и выводит равнозначный ему XML-текст в указанный поток (stream) `PrintWriter`. Код достаточно понятен и помогает проиллюстрировать структуру DOM-дерева. Обратите внимание, что `XMLDocumentWriter` – это только пример. К его недостаткам можно отнести то, что он не обрабатывает все возможные типы узлов DOM и не выводит полное объявление `<!DOCTYPE>`.

Пример 19.4. XMLDocumentWriter.java

```
package com.davidflanagan.examples.xml;
import org.w3c.dom.*;      // классы W3C DOM для навигации по документу
import java.io.*;
```

```
/**
 * Выводим объект DOM Level 1 Document в поток java.io.PrintWriter
 * в виде простого XML-документа. Этот класс не обрабатывает все типы
 * узлов DOM и не разбирается со всеми деталями XML, такими как DTD,
 * символьные кодировки, а также значимые и игнорируемые пробелы.
 * Однако он выводит корректный (well-formed) XML, который может быть
 * разобран непроверяющим (на действительность) анализатором
 **/
public class XMLDocumentWriter {
  PrintWriter out; // Поток для вывода

  /** Инициализируем выходной поток */
  public XMLDocumentWriter(PrintWriter out) { this.out = out; }

  /** Закрываем выходной поток. */
  public void close() { out.close(); }

  /** Выводим в выходной поток узел DOM (например, Document) */
  public void write(Node node) { write(node, ""); }

  /**
   * Выводим указанный объект DOM Node, печатая его
   * с использованием заданной строки отступа
   **/
  public void write(Node node, String indent) {
    // Результат зависит от типа узла
    switch(node.getNodeType()) {
    case Node.DOCUMENT_NODE: {    // Если это узел Document
      Document doc = (Document)node;
      out.println(indent + "<?xml version='1.0'?>"); // Заголовок вывода
      Node child = doc.getFirstChild();  // Берем первый узел
      while(child != null) {             // Цикл до последнего узла
        write(child, indent);            // Выводим узел
        child = child.getNextSibling();  // Получаем следующий узел
      }
      break;
    }
    case Node.DOCUMENT_TYPE_NODE: { // Это тег <!DOCTYPE>
      DocumentType doctype = (DocumentType) node;
      // Заметьте, что DOM Level 1 не дает нам информацию
      // об открытых и системных идентификаторах DOCTYPE,
      // поэтому мы не можем вывести здесь тег <!DOCTYPE>
      // целиком. Мы могли бы сделать это в Level 2.
      out.println("<!DOCTYPE " + doctype.getName() + ">");
      break;
    }
    case Node.ELEMENT_NODE: {    // Большинство узлов - Element
      Element elt = (Element) node;
      out.print(indent + "<" + elt.getTagName());  // Начинаем тег
      NamedNodeMap attrs = elt.getAttributes();    // Получаем атрибуты
      for(int i = 0; i < attrs.getLength(); i++) { // Цикл по ним
        Node a = attrs.item(i);
        out.print(" " + a.getNodeName() + "='" +   // Выводим имя
```

```
            fixup(a.getNodeValue()) + "'");       // Выводим значение
        }
        out.println(">");                  // Завершаем открывающий тег

        String newindent = indent + "  ";  // Увеличиваем отступ
        Node child = elt.getFirstChild();  // Получаем дочерний элемент
        while(child != null) {             // Цикл
          write(child, newindent);         // Выводим элемент
          child = child.getNextSibling();  // Получаем следующий
        }

        out.println(indent + "</" +        // Выводим завершающий тег
              elt.getTagName() + ">");
        break;
      }
      case Node.TEXT_NODE: {               // Текстовый узел
        Text textNode = (Text)node;
        String text = textNode.getData().trim();  // Обрезаем пробелы
        if ((text != null) && text.length() > 0)  // Если непустой,
          out.println(indent + fixup(text));      // печатаем текст
        break;
      }
      case Node.PROCESSING_INSTRUCTION_NODE: {  // Обработка PI-узла
        ProcessingInstruction pi = (ProcessingInstruction)node;
        out.println(indent + "<?" + pi.getTarget() +
                  " " + pi.getData() + "?>");
        break;
      }
      case Node.ENTITY_REFERENCE_NODE: {        // Обработка вставок
        out.println(indent + "&" + node.getNodeName() + ";");
        break;
      }
      case Node.CDATA_SECTION_NODE: {           // Выводим разделы CDATA
        CDATASection cdata = (CDATASection)node;
        // Внимание! Не выводим в программе раздел CDATA непосредственно
        out.println(indent + "<" + "![CDATA[" + cdata.getData() +
              "]]" + ">");
        break;
      }
      case Node.COMMENT_NODE: {         // Комментарии
        Comment c = (Comment)node;
        out.println(indent + "<!--" + c.getData() + "-->");
        break;
      }
      default:  // Надеемся, что это не будет случаться слишком часто!
        System.err.println("Пропускаем узел: " + node.getClass().getName());
        break;
      }
    }

    // Этот метод заменяет зарезервированные символы символьными вставками.
    String fixup(String s) {
```

```
        StringBuffer sb = new StringBuffer();
        int len = s.length();
        for(int i = 0; i < len; i++) {
          char c = s.charAt(i);
          switch(c) {
          default: sb.append(c); break;
          case '<': sb.append("&lt;"); break;
          case '>': sb.append("&gt;"); break;
          case '&': sb.append("&amp;"); break;
          case '"': sb.append("&quot;"); break;
          case '\'': sb.append("&apos;"); break;
          }
        }
        return sb.toString();
      }
    }
```

Навигация по документу с помощью DOM Level 2

В примере 19.5 содержится листинг программы *DOMTreeWalkerTreeModel.java*, демонстрирующей навигацию по дереву DOM с помощью класса `TreeWalker` DOM Level 2. `TreeWalker` является частью пакета `org.w3c.dom.traversal`. Он позволяет вам обходить дерево DOM, используя для этого простой программный интерфейс. Однако более важно то, что он позволяет указать, какие узлы вам нужны, и автоматически отфильтрует все остальные. Он даже разрешает вам создать класс `NodeFilter`, который будет фильтровать узлы, основываясь на любых требуемых вам критериях.

`DOMTreeWalkerTreeModel` реализует интерфейс `javax.swing.tree.TreeModel`, который позволяет вам легко отобразить отфильтрованное дерево DOM при помощи Swing-компонента `JTree`. На рис. 19.1 показан отфильтрованный файл *web.xml*, выведенный таким способом. Но здесь интересны не сами методы `TreeModel` (см. описание `TreeModel` в главе 10), а то, как в реализациях этих методов для навигации по дереву DOM используется `TreeWalker` API.

Метод `main()` анализирует XML-документ, указанный в командной строке, а затем создает для дерева объект `TreeWalker`. `TreeWalker` конфигурируется для показа всех узлов, за исключением комментариев и текстовых узлов, содержащих только пробелы. Затем метод `main()` создает для `TreeWalker` объект `DOMTreeWalkerTreeModel`. В заключение он создает компонент `JTree` для отображения дерева, описанного объектом `DOMTreeWalkerTreeModel`.

Обратите внимание, что этот пример использует анализатор Xerces благодаря наличию в нем поддержки DOM Level 2 (который на момент написания книги не поддерживался JAXP). Так как в примере ис-

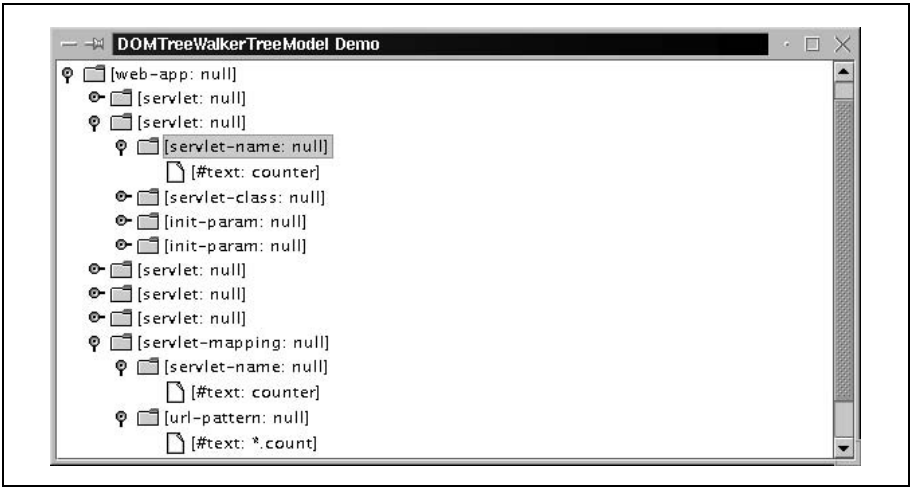


Рис. 19.1. *DOMTreeWalkerTreeModel отображает файл web.xml*

пользуется анализатор Xerces, то для того чтобы откомпилировать и выполнить его, вы должны указать файл *xerces.jar* в переменной окружения `CLASSPATH`. На момент написания книги DOM Level 2 уже был достаточно стабилен, но еще не являлся официальным стандартом. Если `TreeWalker` API изменится в течение этого процесса стандартизации, возможно, это приведет к неверной работе данного примера.

Пример 19.5. DOMTreeWalkerTreeModel.java

```
package com.davidflanagan.examples.xml;
import org.w3c.dom.*;                    // Основные классы DOM
import org.w3c.dom.traversal.*;    // TreeWalker и связанные с ним классы DOM
import org.apache.xerces.parsers.*;   // Классы анализатора Apache Xerces
import org.xml.sax.*;     // Анализатор Xerces DOM использует классы SAX
import javax.swing.*;                    // Классы Swing
import javax.swing.tree.*;               // TreeModel и связанные с ним классы
import javax.swing.event.*;              // Классы событий дерева
import java.io.*;                        // Для чтения входного XML-файла

/**
 * Этот класс реализует интерфейс Swing TreeModel, поэтому дерево DOM,
 * возвращаемое объектом TreeWalker, может быть показано в компоненте JTree.
 **/
public class DOMTreeWalkerTreeModel implements TreeModel {
  TreeWalker walker; // Объект TreeWalker, которым мы моделируем JTree

  /** Создаем TreeModel для указанного объекта TreeWalker */
  public DOMTreeWalkerTreeModel(TreeWalker walker) { this.walker = walker; }

  /**
   * Создаем TreeModel для объекта TreeWalker, который возвращает
   * все узлы указанного документа
   **/
```

```
public DOMTreeWalkerTreeModel(Document document) {
  DocumentTraversal dt = (DocumentTraversal)document;
  walker = dt.createTreeWalker(document, NodeFilter.SHOW_ALL,null,false);
}

/**
 * Создаем TreeModel для объекта TreeWalker, который возвращает
 * указанный элемент и все его подэлементы.
 **/
public DOMTreeWalkerTreeModel(Element element) {
  DocumentTraversal dt = (DocumentTraversal)element.getOwnerDocument();
  walker = dt.createTreeWalker(element, NodeFilter.SHOW_ALL, null,false);
}

// Возвращаем корень дерева
public Object getRoot() { return walker.getRoot(); }

// Является ли узел листом? (Узлы-листья компонент Jtree
// отображает по-другому)
public boolean isLeaf(Object node) {
  walker.setCurrentNode((Node)node); // Задаем текущий узел
  Node child = walker.firstChild();  // Запрашиваем дочерние узлы
  return (child == null);            // Есть ли они у него?
}

// Сколько дочерних элементов у этого узла?
public int getChildCount(Object node) {
  walker.setCurrentNode((Node)node);  // Задаем текущий узел
  // TreeWalker не будет подсчитывать за нас дочерние
  // элементы, поэтому считаем сами
  int numkids = 0;
  Node child = walker.firstChild();  // Начинаем с первого
  while(child != null) {             // Цикл до последнего
    numkids++;                       // Наращиваем счетчик
    child = walker.nextSibling();    // Получаем следующий
  }
  return numkids;    // Это количество подэлементов
}

// Возвращаем указанный подэлемент родительского узла.
public Object getChild(Object parent, int index) {
  walker.setCurrentNode((Node)parent); // Задаем текущий узел
  // TreeWalker предоставляет последовательный, а не произвольный доступ
  // к своим элементам, поэтому мы просматриваем в цикле все элементы
  // один за другим.
  Node child = walker.firstChild();
  while(index-- > 0) child = walker.nextSibling();
  return child;
}

// Возвращаем индекс дочернего узла в родительском узле
public int getIndexOfChild(Object parent, Object child) {
  walker.setCurrentNode((Node)parent);  // Задаем текущий узел
  int index = 0;
```

```
    Node c = walker.firstChild();          // Начинаем с первого
    while((c != child) && (c != null)) {  // Цикл, пока не найдем
      index++;
      c = walker.nextSibling();            // Берем следующий    }
    return index;            // Возвращаем индекс совпадения
  }

  // Требуется только для редактируемых деревьев; здесь не реализован.
  public void valueForPathChanged(TreePath path, Object newvalue) {}

  // Этот объект TreeModel не активизирует никаких событий
  // (из-за того, что дерево нередактируемое), поэтому методы
  // регистрации слушателей событий оставлены нереализованными
  public void addTreeModelListener(TreeModelListener l) {}
  public void removeTreeModelListener(TreeModelListener l) {}

  /**
   * Этот метод main() демонстрирует использование этого
   * класса, использование анализатора Xerces DOM и создание
   * объекта TreeWalker DOM Level 2.
   **/
  public static void main(String[] args) throws IOException, SAXException {
    // Получаем экземпляр анализатора Xerces, необходимый
    // для построения дерева DOM. Заметим, что мы здесь
    // не используем JAXP API, так как этот код использует
    // Apache Xerces API, который не является стандартным
    DOMParser parser = new org.apache.xerces.parsers.DOMParser();

    // Получаем java.io.Reader для входного XML-файла
    // и передаем его во входной источник SAX
    Reader in = new BufferedReader(new FileReader(args[0]));
    InputSource input = new org.xml.sax.InputSource(in);

    // Указываем анализатору Xerces выполнить разбор входного источника
    parser.parse(input);

    // Просим анализатор дать нам наш документ DOM. После того как
    // мы получили дерево DOM, нам больше не нужно использовать
    // Apache Xerces API. С этого момента мы будем использовать
    // стандартный DOM API
    Document document = parser.getDocument();

    // Если мы используем реализацию DOM Level 2, то наш
    // объект Document способен реализовать DocumentTraversal
    DocumentTraversal traversal = (DocumentTraversal)document;

    // Для этого примера мы создадим NodeFilter, который
    // отфильтровывает текстовые узлы, содержащие одни
    // пробелы; они только загромождают дерево
    NodeFilter filter = new NodeFilter() {
        public short acceptNode(Node n) {
          if (n.getNodeType() == Node.TEXT_NODE) {
            // Используем trim() для обрезки начальных и концевых пробелов.
            // Если ничего не остается – отбрасываем узел.
```

```
            if (((Text)n).getData().trim().length() == 0)
              return NodeFilter.FILTER_REJECT;
          }
          return NodeFilter.FILTER_ACCEPT;
        }
      };

    // Этот набор флагов говорит, что нужно показать
    // все типы узлов, за исключением комментариев
    int whatToShow = NodeFilter.SHOW_ALL & ~NodeFilter.SHOW_COMMENT;

    // Создаем TreeWalker с использованием фильтра и флагов
    TreeWalker walker = traversal.createTreeWalker(document, whatToShow,
                                filter, false);

    // Создаем TreeModel и JTree для его отображения
    JTree tree = new JTree(new DOMTreeWalkerTreeModel(walker));

    // Создаем окно с панелью для показа дерева и выводим его
    JFrame frame = new JFrame("DOMTreeWalkerTreeModel Demo");
    frame.getContentPane().add(new JScrollPane(tree));
    frame.setSize(500, 250);
    frame.setVisible(true);
  }
}
```

JDOM API

До этого момента в главе рассматривались официальные, стандартные способы анализа и обработки XML-документов: DOM является стандартом W3C, а SAX стал стандартом де-факто, но так и не был стандартизирован. Однако и SAX, и DOM были предназначены стать программными интерфейсами, независимыми от языков программирования. Однако эта их универсальность означает, что они не могут полностью использовать все особенности языка и платформы Java. Ко времени написания этой главы появился новый (еще в бета-версии), но очень перспективный API, ориентированный непосредственно на Java-программистов. Как видно из его названия, JDOM является объектной моделью XML-документа для Java. Как и DOM API, он создает дерево, представляющее XML-документ. Однако, в отличие от DOM, этот интерфейс с самого начала разрабатывался для Java, и он значительно легче в использовании, чем DOM. JDOM – это проект с открытыми исходными кодами, созданный по инициативе Бретта Мак-Лахлина и Джейсона Хантера (Jason Hunter), авторов книг «Java and XML» и «Java Servlet Programming», соответственно.

В примере 19.6 показано, как можно использовать JDOM API для анализа XML-документа, извлечения информации из полученного дерева, создания новых узловых элементов и добавления их к дереву, а также вывода модифицированного дерева в виде XML-документа.

Сравните этот код с примером 19.3. Эти примеры решают одну и ту же задачу, но, как вы скоро увидите, использование JDOM API делает код более простым и понятным. Вы также заметите, что в JDOM есть собственный встроенный класс `XMLOutputter`, который устраняет необходимость использования класса `XMLDocumentWriter`, показанного в примере 19.4.

Пример 19.6. WebAppConfig2.java

```
package com.davidflanagan.examples.xml;
import java.io.*;
import java.util.*;
import org.jdom.*;
import org.jdom.input.SAXBuilder;
import org.jdom.output.XMLOutputter;

/**
 * Этот класс очень похож на WebAppConfig, но вместо DOM
 * и JAXP API он использует JDOM (Beta 4) API.
 **/
public class WebAppConfig2 {
  /** Метод main()создает и демонстрирует объект WebAppConfig2 */
  public static void main(String[] args)
    throws IOException, JDOMException
  {
    // Создаем новый объект WebAppConfig, представляющий файл
    // web.xml, указанный как первый аргумент командной строки
    WebAppConfig2 config = new WebAppConfig2(new File(args[0]));

    // Запрашиваем у дерева имя класса, ассоциированное с именем сервлета,
    // которое было задано вторым аргументом командной строки
    System.out.println("Класс для сервлета " + args[1] +
         " - " + config.getServletClass(args[1]));

    // Вносим в дерево DOM новое соответствие имя/класс сервлета
    config.addServlet("foo", "bar");

    // И записываем XML-версию дерева DOM в стандартный выходной поток
    config.output(System.out);
  }

  /**
   * Это поле содержит проанализированное дерево JDOM.
   * Заметим, что это документ JDOM, а не DOM.
   **/
  protected org.jdom.Document document;

  /**
   * Считываем заданный файл и анализируем его с целью создания дерева JDOM.
   **/
  public WebAppConfig2(File configfile) throws IOException, JDOMException {
    // JDOM может строить деревья JDOM из различных входных
    // источников. Одним из этих источников является анализатор SAX.
    SAXBuilder builder =
```

```
    new SAXBuilder("org.apache.xerces.parsers.SAXParser");
  // Анализируем заданный файл и преобразуем его в документ JDOM.
  document = builder.build(configfile);
}

/**
 * Этот метод ищет в дереве JDOM заданные узлы Element
 * для того, чтобы определить имя класса, ассоциированное
 * с указанным именем сервлета.
 **/
public String getServletClass(String servletName) throws JDOMException {
  // Получаем корневой элемент документа.
  Element root = document.getRootElement();

  // Находим в документе все элементы <servlet> и в цикле
  // ищем среди них элемент с заданным именем.
  // Обратите внимание на использование класса
  // java.util.List вместо org.w3c.dom.NodeList.
  List servlets = root.getChildren("servlet");
  for(Iterator i = servlets.iterator(); i.hasNext(); ) {
    Element servlet = (Element) i.next();
    // Получаем текст тега <servlet-name>, находящегося
    // внутри тега <servlet>.
    String name = servlet.getChild("servlet-name").getContent();
    if (name.equals(servletName)) {
      // Если имена совпадают, возвращаем текст
      // из элемента <servlet-class>
      return servlet.getChild("servlet-class").getContent();
    }
  }
  return null;
}

/**
 * Этот метод добавляет к документу новое соответствие
 * имя/класс в виде поддерева <servlet>.
 **/
public void addServlet(String servletName, String className)
  throws JDOMException
{
  // Создаем новый элемент, представляющий наш новый сервлет
  Element newServletName = new Element("servlet-name");
  newServletName.setContent(servletName);
  Element newServletClass = new Element("servlet-class");
  newServletClass.setContent(className);
  Element newServlet = new Element("servlet");
  newServlet.addChild(newServletName);
  newServlet.addChild(newServletClass);

  // Находим первый элемент <servlet> документа
  Element root = document.getRootElement();
  Element firstServlet = root.getChild("servlet");
```

```
        // Теперь вставляем новый тег <servlet> перед только что найденным.
        Element parent = firstServlet.getParent();
        List children = parent.getChildren();
        children.add(children.indexOf(firstServlet), newServlet);
      }

      /**
       * Выводим дерево JDOM в виде XML-документа в указанный поток
       **/
      public void output(OutputStream out) throws IOException {
        // JDOM может выводить деревья JDOM различными
        // способами (например, преобразуя в деревья DOM
        // или потоки событий SAX). Здесь мы используем механизм
        // вывода, преобразующий дерево JDOM в XML-документ.
        XMLOutputter outputter = new XMLOutputter(
                        " ",    // выравнивание
                        true);  // используем новые строки
        outputter.output(document, out);
      }
    }
```

Компиляция и выполнение примера

Для того чтобы откомпилировать и выполнить пример 19.6, вам необходимо загрузить дистрибутив JDOM, свободно доступный на сайте *http://www.jdom.org/*. Этот пример был разработан с использованием JDOM версии Beta 4. Из-за бета-статуса JDOM я не буду пытаться давать здесь строгие инструкции по компоновке. Чтобы откомпилировать и выполнить пример, классы JDOM должны быть указаны в вашей переменной `CLASSPATH`. Кроме того, поскольку в примере используется синтаксический анализатор Xerces SAX 2, для выполнения примера в вашей переменной `CLASSPATH` должен присутствовать JAR-файл Xerces. Xerces хорошо состыковывается с JDOM (по крайней мере, с дистрибутивом Beta 4). Наконец, обратите внимание, что JDOM находится в процессе быстрого развития, и его API может несколько измениться со времени выхода используемой здесь версии Beta 4. Если это случится, от вас может потребоваться изменить пример, чтобы он мог компилироваться и выполняться.

Упражнения

19-1. Многие из примеров этой главы были предназначены для анализа файлов *web.xml*, конфигурирующих веб-приложения. Если для запуска ваших сервлетов вы используете контейнер сервлетов Tomcat, то вы, наверное, знаете, что он использует еще один XML-файл *server.xml*, содержащий информацию о конфигурации серверного уровня. В Tomcat 3.1 этот файл находится в каталоге *conf* дистрибутива Tomcat и содержит ряд тегов `<Context>`,

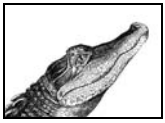
используют атрибуты для задания дополнительной информации по каждому веб-приложению. Напишите программу, которая использует синтаксический анализатор SAX (лучше SAX 2) для разбора файла *server.xml* и выводит значения атрибутов `path` и `docBase` каждого тега `<Context>`.

19-2. Используя вместо анализатора SAX синтаксический анализатор DOM, напишите программу, работающую аналогично программе, которую вы разработали в упражнении 19.1.

19-3. Снова перепишите синтаксический анализатор файла *server.xml*, используя на этот раз JDOM API.

19-4. Напишите основанную на библиотеке Swing программу конфигурирования веб-приложения, которая может читать файлы *web.xml*, разрешать пользователю модифицировать их, а затем сохранять измененную версию. Программа должна предоставлять пользователю возможность добавлять к веб-приложению новые сервлеты и редактировать существующие. Она должна позволять пользователю указать для каждого сервлета его имя, класс, параметры инициализации и шаблон URL.

19-5. Разработайте грамматику XML для представления компонента JavaBeans и значений его свойств. Напишите класс, способный сериализовать произвольный компонент в этот XML-формат и десериализовать, или восстановить, компонент из него обратно. Используйте API отражения Java или класс JavaBeans `Introspector` для идентификации свойств компонента. Примите допущение, что все свойства компонента являются примитивными типами Java, объектами `String` или другими экземплярами компонентов. (Возможно, вам захочется расширить этот список, включив в него объекты `Font` и `Color`.) Далее предположите, что во всех классах компонентов определены конструкторы без аргументов и что все компоненты могут быть должным образом инициализированы путем создания их экземпляров и установки значений их открытых свойств.



Алфавитный указатель

Специальные символы

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

А

Б

В

Г

Д

Ж

З

И

Л

М

Н

O

П

P

С

T

У

Ф

Х

Ц

Ч

Ш

Щ

Э