

ПОЛНОСТЬЮ ОБНОВЛЕНО ПО ВЕРСИИ JAVA SE 8



JavaTM

БИБЛИОТЕКА ПРОФЕССИОНАЛА

Том 2. Расширенные средства
программирования

ДЕСЯТОЕ ИЗДАНИЕ



Кей Хорстманн

Библиотека профессионала

Java[™]

Том 2. Расширенные
средства
программирования

Десятое издание

Core JavaTM

Volume II – Advanced Features

Tenth Edition

Cay S. Horstmann



PRENTICE
HALL

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam
• Cape Town • Dubai • London • Madrid • Milan • Munich • Paris • Paris • Montreal • Toronto
• Delhi • Mexico City • São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Библиотека профессионала

Java[™]

Том 2. Расширенные
средства
программирования

Десятое издание

Кей Хорстманн



Компьютерное издательство "Диалектика"
Москва • Санкт-Петербург • Киев
2017

ББК 32.973.26-018.2.75

Х82

УДК 681.3.07

Компьютерное издательство "Диалектика"

Зав. редакцией С.Н. Тригуб

Перевод с английского и редакция И.В. Берштейна

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:

info@dialektika.com, http://www.dialektika.com

Хорстманн, Кей С.

X82 Java. Библиотека профессионала, том 2. Расширенные средства программирования, 10-е изд. : Пер. с англ. — Спб. : ООО "Альфа-книга", 2017. — 976 с. : ил. — Парал. тит. англ.

ISBN 978-5-9909445-0-3 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

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

Authorized translation from the English language edition published by Prentice Hall, Inc., Copyright © 2017 Oracle and/or its affiliates. All rights reserved.

Portions © 2017 Cay S. Horstmann

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

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2017

Научно-популярное издание

Кей С. Хорстмани

Java. Библиотека профессионала, том 2 Расширенные средства программирования, 10-е издание

Литературный редактор **И.А. Попова**

Верстка **О.В. Мишутина**

Художественный редактор **В.Г. Павлютин**

Корректор **Л.А. Гордиенко**

Подписано в печать 17.07.2017. Формат 70x100/16

Гарнитура Times. Печать офсетная

Усл. печ. л. 78,69. Уч.-изд. л. 56,8

Тираж 500 экз. Заказ № 4904

Отпечатано в АО «Первая Образцовая типография»

Филиал «Чеховский Печатный Двор»

142300, Московская область, г. Чехов, ул. Полиграфистов, д.1

ООО "Альфа-книга", 195027, Санкт-Петербург, Магнитогорская ул., д. 30

ISBN 978-5-9909445-0-3 (рус.)

© Компьютерное издательство "Диалектика", 2017,
перевод, оформление, макетирование

ISBN 978-0-13-417729-8 (англ.)

© 2017 Oracle and/or its affiliates.

Оглавление

Предисловие	13
Глава 1. Библиотека потоков данных в Java SE 8	19
Глава 2. Ввод и вывод	63
Глава 3. XML	153
Глава 4. Работа в сети	235
Глава 5. Работа с базами данных	279
Глава 6. Прикладной программный интерфейс API даты и времени	345
Глава 7. Интернационализация	365
Глава 8. Написание сценариев, компиляция и обработка аннотаций	419
Глава 9. Безопасность	475
Глава 10. Расширенные средства Swing	559
Глава 11. Расширенные средства AWT	733
Глава 12. Платформенно-ориентированные методы	895
Предметный указатель	955

Содержание

Предисловие	13
К читателю	13
Краткий обзор книги	13
Условные обозначения	16
Примеры исходного кода	16
Благодарности	16
От издательства	18
Глава 1. Библиотека потоков данных в Java SE 8	19
1.1. От итерации к потоковым операциям	20
1.2. Создание потока данных	22
1.3. Методы <code>filter()</code> , <code>map()</code> и <code>flatMap()</code>	26
1.4. Извлечение подпотоков и сцепление потоков данных	27
1.5. Другие операции преобразования потоков данных	28
1.6. Простые методы сведения	29
1.7. Тип <code>Optional</code>	30
1.7.1. Как обращаться с необязательными значениями	31
1.7.2. Как не следует обращаться с необязательными значениями	32
1.7.3. Формирование необязательных значений	33
1.7.4. Сочетание функций необязательных значений с методом <code>flatMap()</code>	33
1.8. Накопление результатов	36
1.9. Накопление результатов в отображениях	40
1.10. Группирование и разделение	44
1.11. Нисходящие коллекторы	45
1.12. Операции сведения	49
1.13. Потоки данных примитивных типов	51
1.14. Параллельные потоки данных	57
Глава 2. Ввод и вывод	63
2.1. Потоки ввода-вывода	63
2.1.1. Чтение и запись байтов	64
2.1.2. Полный комплект потоков ввода-вывода	66
2.1.3. Сочетание фильтров потоков ввода-вывода	71
2.2. Ввод-вывод текста	74
2.2.1. Вывод текста	75
2.2.2. Ввод текста	77
2.2.3. Сохранение объектов в текстовом формате	78
2.2.4. Кодировки символов	81
2.3. Чтение и запись двоичных данных	84
2.3.1. Интерфейсы <code>DataInput</code> и <code>DataOutput</code>	84
2.3.2. Файлы с произвольным доступом	87
2.3.3. ZIP-архивы	91

2.4. Потоки ввода-вывода и сериализация объектов	94
2.4.1. Сохранение и загрузка сериализуемых объектов	95
2.4.2. Представление о формате файлов для сериализации объектов	100
2.4.3. Видоизменение исходного механизма сериализации	106
2.4.4. Сериализация одноэлементных множеств и типизированных перечислений	108
2.4.5. Контроль версий	109
2.4.6. Применение сериализации для клонирования	111
2.5. Манипулирование файлами	114
2.5.1. Пути к файлам	114
2.5.2. Чтение и запись данных в файлы	117
2.5.3. Создание файлов и каталогов	118
2.5.4. Копирование, перемещение и удаление файлов	119
2.5.5. Получение сведений о файлах	121
2.5.6. Обход элементов каталога	123
2.5.7. Применение потоков каталогов	124
2.5.8. Системы ZIP-файлов	127
2.6. Файлы, отображаемые в памяти	128
2.6.1. Эффективность файлов, отображаемых в памяти	129
2.6.2. Структура буфера данных	136
2.6.3. Блокирование файлов	138
2.7. Регулярные выражения	141
Глава 3. XML	153
3.1. Введение в XML	154
3.1.1. Структура XML-документа	156
3.2. Синтаксический анализ XML-документов	159
3.3. Проверка достоверности XML-документов	171
3.3.1. Определения типов документов	172
3.3.2. Схема XML-документов	179
3.3.3. Практический пример применения XML-документов	182
3.4. Поиск информации средствами XPath	195
3.5. Использование пространств имен	201
3.6. Потоковые синтаксические анализаторы	203
3.6.1. Применение SAX-анализатора	204
3.6.2. Применение StAX-анализатора	209
3.7. Формирование XML-документов	213
3.7.1. XML-документы без пространств имен	213
3.7.2. XML-документы с пространствами имен	214
3.7.3. Запись XML-документов	214
3.7.4. Пример формирования файла в формате SVG	215
3.7.5. Запись XML-документов средствами StAX	218
3.8. Преобразование XML-документов языковыми средствами XSLT	225
Глава 4. Работа в сети	235
4.1. Подключение к серверу	235
4.1.1. Применение утилиты telnet	235
4.1.2. Подключение к серверу из программы на Java	238
4.1.3. Время ожидания для сокетов	240
4.1.4. Межсетевые адреса	241

4.2. Реализация серверов	243
4.2.1. Сокеты сервера	243
4.2.2. Обслуживание многих клиентов	246
4.2.3. Полузакрытие	250
4.3. Прерываемые сокеты	251
4.4. Получение данных из Интернета	257
4.4.1. URL и URI	257
4.4.2. Извлечение данных средствами класса URLConnection	259
4.4.3. Отправка данных формы	267
4.5. Отправка электронной почты	275
Глава 5. Работа с базами данных	279
5.1. Структура JDBC	280
5.1.1. Типы драйверов JDBC	280
5.1.2. Типичные примеры применения JDBC	282
5.2. Язык SQL	283
5.3. Конфигурирование JDBC	288
5.3.1. URL баз данных	289
5.3.2. Архивные JAR-файлы драйверов	289
5.3.3. Запуск базы данных	289
5.3.4. Регистрация класса драйвера	290
5.3.5. Подключение к базе данных	291
5.4. Работа с операторами JDBC	294
5.4.1. Выполнение команд SQL	294
5.4.2. Управление соединениями, командами и результирующими наборами	297
5.4.3. Анализ исключений SQL	298
5.4.4. Заполнение базы данных	301
5.5. Выполнение запросов	304
5.5.1. Подготовленные операторы и запросы	305
5.5.2. Чтение и запись больших объектов	311
5.5.3. Синтаксис переходов в SQL	313
5.5.4. Множественные результаты	314
5.5.5. Извлечение автоматически генерируемых ключей	315
5.6. Прокручиваемые и обновляемые результирующие наборы	316
5.6.1. Прокручиваемые результирующие наборы	316
5.6.2. Обновляемые результирующие наборы	318
5.7. Наборы строк	323
5.7.1. Создание наборов строк	323
5.7.2. Кешируемые наборы строк	324
5.8. Метаданные	327
5.9. Транзакции	337
5.9.1. Программирование транзакций средствами JDBC	337
5.9.2. Точки сохранения	338
5.9.3. Групповые обновления	338
5.10. Расширенные типы данных SQL	340
5.11. Управление подключением к базам данных в веб-приложениях и производственных приложениях	342

Глава 6. Прикладной программный интерфейс API даты и времени	345
6.1. Временная шкала	346
6.2. Местные даты	349
6.3. Корректоры дат	352
6.4. Местное время	353
6.5. Поясное время	354
6.6. Форматирование и синтаксический анализ даты и времени	358
6.7. Взаимодействие с унаследованным кодом	362
Глава 7. Интернационализация	365
7.1. Региональные настройки	366
7.2. Числовые форматы	371
7.3. Форматирование денежных сумм в разных валютах	377
7.4. Форматирование даты и времени	378
7.5. Сортировка и нормализация	385
7.6. Форматирование сообщений	392
7.6.1. Форматирование чисел и дат	392
7.6.2. Форматы выбора	394
7.7. Ввод-вывод текста	396
7.7.1. Текстовые файлы	397
7.7.2. Окончания строк	397
7.7.3. Консольный ввод-вывод	397
7.7.4. Протокольные файлы	398
7.7.5. Отметка порядка следования байтов в кодировке UTF-8	398
7.7.6. Кодирование символов в исходных файлах	399
7.8. Комплекты ресурсов	400
7.8.1. Обнаружение комплектов ресурсов	400
7.8.2. Файлы свойств	402
7.8.3. Классы комплектов ресурсов	402
7.9. Пример интернационализации прикладной программы	404
Глава 8. Написание сценариев, компиляция и обработка аннотаций	419
8.1. Написание сценариев для платформы Java	420
8.1.1. Получение механизма сценариев	420
8.1.2. Выполнение сценариев и привязки	421
8.1.3. Переадресация ввода-вывода	423
8.1.4. Вызов сценарных функций и методов	424
8.1.5. Компиляция сценариев	426
8.1.6. Пример создания сценария для обработки событий в ГПИ	426
8.2. Прикладной программный интерфейс API для компилятора	431
8.2.1. Простой способ компилирования	432
8.2.2. Выполнение заданий на компиляцию	432
8.2.3. Пример динамического генерирования кода Java	438
8.3. Применение аннотаций	443
8.3.1. Введение в аннотации	443
8.3.2. Пример аннотирования обработчиков событий	445
8.4. Синтаксис аннотаций	450
8.4.1. Интерфейсы аннотаций	450
8.4.2. Объявление аннотаций	451
8.4.3. Аннотирование объявлений	453

8.4.4. Аннотирование в местах употребления типов данных	454
8.4.5. Аннотирование по ссылке <code>this</code>	455
8.5. Стандартные аннотации	456
8.5.1. Аннотации для компиляции	457
8.5.2. Аннотации для управления ресурсами	458
8.5.3. Мета-аннотации	458
8.6. Обработка аннотаций на уровне исходного кода	461
8.6.1. Процессоры аннотаций	461
8.6.2. Прикладной программный интерфейс API модели языка	462
8.6.3. Генерирование исходного кода с помощью аннотаций	463
8.7. Конструирование байт-кодов	466
8.7.1. Видоизменение файлов классов	466
8.7.2. Видоизменение байт-кодов во время загрузки	471
Глава 9. Безопасность	475
9.1. Загрузчики классов	476
9.1.1. Процесс загрузки классов	476
9.1.2. Иерархия загрузчиков классов	477
9.1.3. Применение загрузчиков классов в качестве пространств имен	480
9.1.4. Создание собственного загрузчика классов	481
9.1.5. Верификация байт-кода	486
9.2. Диспетчеры защиты и полномочия	490
9.2.1. Проверка полномочий	491
9.2.2. Организация защиты на платформе Java	493
9.2.3. Файлы правил защиты	496
9.2.4. Пользовательские полномочия	503
9.2.5. Реализация класса полномочий	504
9.3. Аутентификация пользователей	510
9.3.1. Каркас JAAS	510
9.3.2. Модули регистрации JAAS	516
9.4. Цифровые подписи	525
9.4.1. Свертки сообщений	526
9.4.2. Подписание сообщений	529
9.4.3. Верификация подписи	530
9.4.4. Проблема аутентификации	534
9.4.5. Подписание сертификатов	537
9.4.6. Запросы сертификатов	538
9.4.7. Подписание кода	539
9.5. Шифрование	545
9.5.1. Симметричные шифры	546
9.5.2. Генерирование ключей шифрования	547
9.5.3. Потоки шифрования	552
9.5.4. Шифрование открытым ключом	553
Глава 10. Расширенные средства Swing	559
10.1. Списки	560
10.1.1. Компонент <code>JList</code>	560
10.1.2. Модели списков	566
10.1.3. Ввод и удаление значений	571
10.1.4. Воспроизведение значений	572

10.2. Таблицы	576
10.2.1. Простая таблица	577
10.2.2. Модели таблиц	581
10.2.3. Манипулирование строками и столбцами таблицы	585
10.2.4. Воспроизведение и редактирование ячеек	601
10.3. Деревья	614
10.3.1. Простые деревья	615
10.3.3. Перечисление узлов дерева	632
10.3.4. Воспроизведение узлов дерева	634
10.3.5. Обработка событий в деревьях	637
10.3.6. Специальные модели деревьев	644
10.4. Текстовые компоненты	652
10.4.1. Отслеживание изменений в текстовых компонентах	653
10.4.2. Поля ввода форматируемого текста	657
10.4.3. Компонент JSpinner	674
10.4.4. Отображение HTML-документов средствами JEditorPane	682
10.5. Индикаторы состояния	688
10.5.1. Индикаторы выполнения	688
10.5.2. Мониторы текущего состояния	692
10.5.3. Контроль процесса чтения данных из потока ввода	695
10.6. Организаторы и декораторы компонентов	700
10.6.1. Разделяемые панели	701
10.6.2. Панели с вкладками	704
10.6.3. Настольные панели и внутренние фреймы	710
10.6.4. Слои	728
Глава 11. Расширенные средства AWT	733
11.1. Конвейер визуализации	734
11.2. Фигуры	736
11.2.1. Иерархия классов рисования фигур	737
11.2.2. Применение классов рисования фигур	738
11.3. Участки	752
11.4. Обводка	754
11.5. Раскраска	762
11.6. Преобразование координат	764
11.7. Отсечение	770
11.8. Прозрачность и композиция	772
11.9. Указания по воспроизведению	781
11.10. Чтение и запись изображений	787
11.10.1. Получение средств чтения	
и записи изображений по типам файлов	788
11.10.2. Чтение и запись файлов с несколькими изображениями	789
11.11. Манипулирование изображениями	797
11.11.1. Формирование растровых изображений	798
11.11.2. Фильтрация изображений	805
11.12. Вывод изображений на печать	813
11.12.1. Вывод двухмерной графики на печать	814
11.12.2. Многостраницчная печать	823
11.12.3. Предварительный просмотр печати	825
11.12.4. Службы печати	834
11.12.5. Потоковые службы печати	838

11.12.6. Атрибуты печати	839
11.13. Буфер обмена	846
11.13.1. Классы и интерфейсы для передачи данных	847
11.13.2. Передача текста	847
11.13.3. Интерфейс Transferable и разновидности данных	851
11.13.4. Передача изображений через буфер обмена	853
11.13.5. Передача объектов Java через системный буфер обмена	857
11.13.6. Передача ссылок на объекты через локальный буфер обмена	860
11.14. Перетаскивание объектов	861
11.14.1. Поддержка передачи данных в Swing	863
11.14.2. Источники перетаскивания	867
11.14.3. Приемники перетаскивания	870
11.15. Интеграция с платформой	878
11.15.1. Начальные экраны	879
11.15.2. Запуск настольных приложений	884
11.15.3. Системная область	889
Глава 12. Платформенно-ориентированные методы	895
12.1 Вызов функции на C из программы на Java	896
12.2. Числовые параметры и возвращаемые значения	902
12.3. Строковые параметры	904
12.4. Доступ к полям	910
12.4.1. Доступ к полям экземпляра	911
12.4.2. Доступ к статическим полям	914
12.5. Кодирование сигнатур	915
12.6. Вызов методов на Java	917
12.6.1. Методы экземпляра	917
12.6.2. Статические методы	919
12.6.3. Конструкторы	920
12.6.4. Альтернативные вызовы методов	920
12.7. Доступ к элементам массивов	924
12.8. Обработка ошибок	928
12.9. Применение прикладного программного интерфейса API для вызовов	933
12.10. Пример обращения к реестру Windows	938
12.10.1. Общее представление о реестре Windows	938
12.10.2. Интерфейс для доступа к реестру на платформе Java	940
12.10.3. Реализация функций доступа к реестру в виде платформенно-ориентированных методов	941
Предметный указатель	955

Предисловие

К читателю

Вы держите в руках второй том десятого издания, полностью обновленного по версии Java SE 8. В первом томе рассматривались основные языковые средства Java, а в этом томе речь пойдет о расширенных функциональных возможностях, которые могут понадобиться программисту для разработки программного обеспечения на высоком профессиональном уровне. Поэтому этот том, как, впрочем, и первый том настоящего и предыдущих изданий данной книги, нацелен на тех программистов, которые собираются применять технологию Java в реальных проектах.

Краткий обзор книги

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

В **главе 1** рассматривается библиотека потоков данных, внедренная в версии Java 8 и придающая современные черты обработке данных благодаря тому, что программисту достаточно указать, что именно ему требуется, не вдаваясь в подробности, как получить желаемый результат. Такой подход позволяет уделить основное внимание в библиотеке потоков данных оптимальной эволюционной стратегии, которая дает особые преимущества при оптимизации параллельных вычислений.

Глава 2 посвящена организации ввода-вывода. В Java весь ввод-вывод осуществляется через так называемые *потоки ввода-вывода* (не путать с потоками данных, рассматриваемыми в главе 1). Такие потоки позволяют единообразно обмениваться данными между различными источниками, включая файлы, сетевые соединения и блоки памяти. В начале этой главы подробно описаны классы чтения и записи в потоки ввода-вывода, упрощающие обработку данных в Юникоде. Затем рассматривается внутренний механизм сериализации объектов, который упрощает сохранение и загрузку объектов. И в завершение главы обсуждаются регулярные выражения, а также особенности манипулирования файлами и путями к ним.

Основной темой **главы 3** является XML. В ней показано, каким образом осуществляется синтаксический анализ XML-файлов, генерируется разметка в коде XML и выполняются XSL-преобразования. В качестве примера демонстрируется порядок

обозначения компоновки Swing-формы в формате XML. В этой главе рассматривается также прикладной программный интерфейс API XPath, в значительной степени упрощающий поиск мелких подробностей в больших объемах данных XML.

В главе 4 рассматривается сетевой прикладной программный интерфейс API. В языке Java чрезвычайно просто решаются сложные задачи сетевого программирования. И в этой главе будет показано, как устанавливаются сетевые соединения с серверами, реализуются собственные серверы и организуется связь по протоколу HTTP.

Глава 5 посвящена программированию баз данных. Основное внимание в ней уделяется интерфейсу JDBC (прикладному программному интерфейсу API для организации доступа к базам данных из приложений на Java), который позволяет прикладным программам на Java устанавливать связь с реляционными базами данных. Будет также показано, как писать полезные программы для выполнения рутинных операций с настоящими базами данных, применяя только базовое подмножество интерфейса JDBC. (Для рассмотрения всех средств интерфейса JDBC потребовалась бы отдельная книга почти такого же объема, как и эта.) И в завершение главы приводятся краткие сведения об интерфейсе JNDI (Java Naming and Directory Interface — интерфейс именования и каталогов Java) и протоколе LDAP (Lightweight Directory Access Protocol — упрощенный каталог доступа к каталогам).

Ранее в библиотеках Java были предприняты две безуспешные попытки организовать обработку даты и времени. Третья попытка была успешно предпринята в версии Java 8. Поэтому в главе 6 поясняется, как преодолевать трудности организации календарей и манипулирования часовыми поясами, используя новую библиотеку даты и времени.

В главе 7 обсуждаются вопросы интернационализации, важность которой, на наш взгляд, будет со временем только возрастать. Java относится к тем немногочисленным языкам программирования, в которых с самого начала предусматривалась возможность обработки данных в Юникоде, но поддержка интернационализации в Java этим не ограничивается. В частности, интернационализация прикладных программ на Java позволяет сделать их независимыми не только от платформы, но и от страны применения. В качестве примера в этой главе демонстрируется, как написать прикладную программу для расчета времени выхода на пенсию с выбором английского, немецкого или китайского языка, в зависимости от региональных настроек в браузере.

В главе 8 описываются три технологии для обработки кода. Так, прикладные программные интерфейсы API для сценариев и компилятора дают возможность вызывать в программе на Java код, написанный на каком-нибудь языке создания сценариев, например JavaScript или Groovy, и компилировать его в код Java. Аннотации позволяют вводить в программу на Java произвольно выбираемую информацию (иногда еще называемую *метаданными*). В этой главе показывается, каким образом обработчики аннотаций собирают аннотации на уровне источника и на уровне файлов классов и как с помощью аннотаций оказывается воздействие на поведение классов во время выполнения. Аннотации выгодно использовать только вместе с подходящими инструментальными средствами, и мы надеемся, что материал этой главы поможет читателю научиться выбирать именно те средства обработки аннотаций, которые в наибольшей степени отвечают его потребностям.

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

В главе 10 представлен весь материал по библиотеке Swing, который не вошел в первый том данной книги, в том числе описание важных и сложных компонентов деревьев и таблиц. В ней демонстрируются основные способы применения панелей редактора, реализация в Java многодокументного интерфейса, индикаторы выполнения, применяемые в многопоточных программах, а также средства интеграции для рабочего стола вроде заставок и поддержки области уведомлений на панели задач. И здесь основное внимание уделяется только наиболее полезным конструкциям, которые разработчикам чаще всего приходится использовать на практике, поскольку даже краткое рассмотрение всей библиотеки Swing заняло бы не один том и представляло бы интерес только для очень узких специалистов.

В главе 11 рассматривается прикладной программный интерфейс Java 2D API, которым можно пользоваться для рисования реалистичных графических изображений и спецэффектов. В ней также рассказывается о некоторых более развитых средствах библиотеки AWT (Abstract Windowing Toolkit — набор инструментальных средств для абстрактных окон), которые оказались слишком специфическими для рассмотрения в первом томе и, тем не менее, должны быть включены в арсенал средств всякого программирующего на Java. К их числу относятся средства вывода на печать и прикладные программные интерфейсы API, позволяющие выполнять операции вырезания, вставки и перетаскивания объектов.

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

Как обычно, все главы второго тома были полностью обновлены по самой последней версии Java. Весь устаревший материал был изъят, а новые прикладные программные интерфейсы API, появившиеся в версии Java SE 8, подробно рассматриваются в соответствующих местах.

Условные обозначения

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



Этой пиктограммой выделяются примечания.



Этой пиктограммой выделяются советы.



Этой пиктограммой выделяются предупреждения о потенциальной опасности.



В настоящей книге имеется немало примечаний к синтаксису C++, где разъясняются отличия между языками Java и C++. Можете пропустить их, если вас не интересует программирование на C++.

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

Название прикладного программного интерфейса **1.2**

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

Листинг 1.1. Исходный код из файла `ScriptTest.java`

Примеры исходного кода

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

Благодарности

Написание книги всегда требует значительных усилий, а ее переписывание не намного легче, особенно если учесть постоянные изменения в технологии Java.

Чтобы сделать книгу полезной, необходимы совместные усилия многих преданных людей, и автор с удовольствием выражает признательность всем, кто внес свой посильный вклад в общее дело.

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

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

Среди рецензентов этого и предыдущих изданий хотелось бы отметить Чака Аллисона (Chuck Allison, выпускающего редактора издания *C/C++ Users Journal*), Ланса Андерсона (Lance Anderson) из компании Oracle, Алекса Битона (Alec Beaton) из PointBase, Inc., Клиффа Берга (Cliff Berg), Джошуа Блоха (Joshua Bloch), Дэвида Брауна (David Brown), Корки Карtrightа (Corky Cartwright), Френка Коена (Frank Cohen) из PushToTest, Криса Крейна (Chris Crane) из devXsolution, доктора Николаса Дж. Де Лилло (Dr. Nicholas J. De Lillo) из Манхэттенского колледжа, Ракеша Дхупара (Rakesh Dhoopar) из компании Oracle, Роберта Эванса (Robert Evans), ведущего специалиста из лаборатории прикладной физики университета имени Джонса Хопкинса, Дэвида Джирри (David Garry) из Sabreware, Джима Гиша (Jim Gish) из Oracle, Брайана Гоетца (Brian Goetz) из Oracle, Анджелу Гордон (Angela Gordon), Дэна Гордона (Dan Gordon), Роба Гордона (Rob Gordon), Джона Грэя (John Gray) из Хартфордского университета, Камерона Грегори (Cameron Gregory, olabs.com), Марти Холла (Marty Hall) из лаборатории прикладной физики в университете имени Джона Хопкинса, Винсента Харди (Vincent Hardy) из Adobe Systems, Дэна Харки (Dan Harkey) из университета штата Калифорния в Сан-Хоце, Вильяма Хиггинса (William Higgins) из IBM, Владимира Ивановича (Vladimir Ivanovic) из PointBase, Джерри Джексона (Jerry Jackson) из CA Technologies, Тима Киммета (Tim Kimmel) из Walmart, Криса Лаффра (Chris Laffra), Чарли Лай (Charlie Lai) из компании Apple, Анжелику Лангер (Angelika Langer), Дуга Лэнгстона (Doug Langston), Ханг Лай (Hang Lau) из университета имени Макгилла, Марка Лоуренса (Mark Lawrence), Дуга Ли (Doug Lea) из SUNY Oswego, Грегори Лонгшора (Gregory Longshore), Боба Линча (Bob Lynch) из Lynch Associates, Филиппа Милна (Philip Milne), консультанта, Марка Моррисси (Mark Morrissey) из научно-исследовательского института штата Орегон, Махеш Нилаканта (Mahesh Neelakanta) из Атлантического университета штата Флорида, Хао Фам (Hao Pham), Пола Филиона (Paul Philion), Блейка Рагсдейла (Blake Ragsdell), Ильбера Рамадани (Ylber Ramadani) из университета имени Райерсона, Стюарта Реджеса (Stuart Reges) из университета штата Аризона, Саймона Риттера (Simon

Ritter), Рича Розена (Rich Rosen) из Interactive Data Corporation, Питера Сандерса (Peter Sanders) из университета ЭССИ (ESSI), г. Ницца, Франция, доктора Пола Сангерера (Dr. Paul Sanghera) из университета штата Калифорния в Сан-Хосе и колледжа имени Брукса, Поля Севинка (Paul Sevinc) из Teamip AG, Деванга Шаха (Devang Shah), Йокиси Сабата (Yoshiki Shabata), Ричарда Сливчака (Richard Slywczak) из Исследовательского центра имени Гленна, НАСА, Бредли А. Смита (Bradley A. Smith), Стивена Стелтинга (Steven Stelting), Кристофера Тэйлора (Christopher Taylor), Люка Тэйлора (Luke Taylor) из Valtech, Джорджа Тхируваткуала (George Thiruvathukal), Кима Топли (Kim Topley), автора книги *Core JFC, Second Edition*, Джанет Трауб (Janet Traub), Пола Тайму (Paul Tuma), консультанта, Кристиана Улленбоома (Christian Ullenboom), Питера Ван Дер Линдена (Peter van der Linden) из Motorola Mobile Devices, Берта Уолша (Burt Walsh), Джо Уанга (Joe Wang) из Oracle и Дана Ксю (Dan Xu) из Oracle.

Кей Хорстманн, Сан-Франциско, сентябрь 2016 г.

От издательства

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

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

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

Наши электронные адреса:

E-mail: info@dialektika.com

WWW: <http://www.dialektika.com>

Наши почтовые адреса:

в России: 195027, Санкт-Петербург, Магнитогорская ул., д. 30, ящик 116

в Украине: 03150, Киев, а/я 152

Библиотека потоков данных в Java SE 8

В этой главе...

- ▶ От итерации к потоковым операциям
- ▶ Создание потока данных
- ▶ Методы `filter()`, `map()` и `flatMap()`
- ▶ Извлечение подпотоков и сцепление потоков данных
- ▶ Другие операции преобразования потоков данных
- ▶ Простые методы сведения
- ▶ Тип `Optional`
- ▶ Накопление результатов
- ▶ Накопление результатов в отображениях
- ▶ Группирование и разделение
- ▶ Нисходящие коллекторы
- ▶ Операции сведения
- ▶ Потоки данных примитивных типов
- ▶ Параллельные потоки данных

Потоки данных обеспечивают представление данных, позволяющее указать вычисления на более высоком концептуальном уровне, чем коллекции. С помощью потока данных можно указать, что и как именно требуется сделать с данными, а планирование операций предоставить конкретной реализации. Допустим,

требуется вычислить среднее некоторого свойства. С этой целью указывается источник данных и свойство, а средствами библиотеки потоков данных можно оптимизировать вычисление, используя, например, несколько потоков исполнения для расчета сумм, подсчета и объединения результатов.

В этой главе поясняется, как пользоваться библиотекой потоков данных, которая была внедрена в версии Java SE 8, для обработки коллекций по принципу “что, а не как делать”.

1.1. От итерации к потоковым операциям

Для обработки коллекции обычно требуется перебрать ее элементы и выполнить над ними некоторую операцию. Допустим, требуется подсчитать все длинные слова в книге. Сначала организуем их вывод списком:

```
String contents = new String(Files.readAllBytes(  
    Paths.get("alice.txt")), StandardCharsets.UTF_8);  
    // прочитать текст из файла в символьную строку  
List<String> words = Arrays.asList(contents.split("\\PL+"));  
    // разбить полученную символьную строку на слова;  
    // небуквенные символы считаются разделителями
```

А теперь можно перебрать слова таким образом:

```
int count = 0;  
for (String w : words) {  
    if (w.length() > 12) count++;  
}
```

Ниже показано, как аналогичная операция осуществляется с помощью потоков данных.

```
long count = words.stream()  
    .filter(w -> w.length() > 12)  
    .count();
```

В последнем случае не нужно искать в цикле наглядного подтверждения операций фильтрации и подсчета слов. Сами имена методов свидетельствуют о том, что именно предполагается сделать в коде. Более того, если в цикле во всех подробностях предписывается порядок выполнения операций, то в потоке данных операции можно планировать как угодно, при условии, что будет достигнут правильный результат.

Достаточно заменить метод `stream()` на метод `parallelStream()`, чтобы организовать средствами библиотеки потоков данных параллельное выполнение операций фильтрации и подсчета слов:

```
long count = words.parallelStream()  
    .filter(w -> w.length() > 12)  
    .count();
```

Потоки данных действуют по принципу “что, а не как делать”. В рассматриваемом здесь примере кода мы описываем, что нужно сделать: получить длинные слова и подсчитать их. При этом мы не указываем, в каком порядке или потоке исполнения это должно произойти. Напротив, в упомянутом выше цикле точно

указывается порядок организации вычислений, а следовательно, исключается всякая возможность для оптимизации.

На первый взгляд поток данных похож на коллекцию, поскольку он позволяет преобразовывать и извлекать данные. Но у потока данных имеются следующие существенные отличия.

1. Поток данных не сохраняет свои элементы. Они могут храниться в основной коллекции или формироваться по требованию.
2. Потоковые операции не изменяют их источник. Например, метод `filter()` не удаляет элементы из нового потока данных, но выдает новый поток, в котором они отсутствуют.
3. Потоковые операции выполняются *по требованию*, когда это возможно. Это означает, что они не выполняются до тех пор, пока не потребуется их результат. Так, если требуется подсчитать только пять длинных слов вместо всех слов, метод `filter()` прекратит фильтрацию после пятого совпадения. Следовательно, потоки данных могут быть бесконечными!

Вернемся к предыдущему примеру, чтобы рассмотреть его подробнее. Методы `stream()` и `parallelStream()` выдают поток данных для списка слов `words`. А метод `filter()` возвращает другой поток данных, содержащий только те слова, длина которых больше 12 букв. И наконец, метод `count()` сводит этот поток данных в конечный результат.

Такая последовательность операций весьма характерна для манипулирования потоками данных. Конвейер операций организуется в следующие три стадии.

1. Создание потока данных.
2. Указание *промежуточных операций* для преобразования исходного потока данных в другие потоки — возможно, в несколько этапов.
3. Выполнение *окончечной операции* для получения результата. Эта операция принуждает к выполнению по требованию тех операций, которые ей предшествуют. А впоследствии поток данных может больше не понадобиться.

В примере кода из листинга 1.1 поток данных создается методом `stream()` или `parallelStream()`. Метод `filter()` преобразует его, а метод `count()` выполняет окончечную операцию.

Листинг 1.1. Исходный код из файла streams/CountLongWords.java

```
1 package streams;
2
3 import java.io.IOException;
4 import java.nio.charset.StandardCharsets;
5 import java.nio.file.Files;
6 import java.nio.file.Paths;
7 import java.util.Arrays;
8 import java.util.List;
9
10 public class CountLongWords
11 {
12     public static void main(String[] args) throws IOException
```

```

13  {
14      String contents = new String(Files.readAllBytes(Paths.get(
15          "../gutenberg/alice30.txt")), StandardCharsets.UTF_8);
16      List<String> words = Arrays.asList(contents.split("\\PL+"));
17
18      long count = 0;
19      for (String w : words)
20      {
21          if (w.length() > 12) count++;
22      }
23      System.out.println(count);
24
25      count = words.stream().filter(w -> w.length() > 12).count();
26      System.out.println(count);
27
28      count =
29          words.parallelStream().filter(w -> w.length() > 12).count();
30      System.out.println(count);
31  }
32 }
```

В следующем разделе будет показано, как создается поток данных. В трех последующих разделах рассматриваются потоковые операции преобразования, а в пяти следующих за ними разделах — окончные операции.

`java.util.stream.Stream<T>` 8

- **`Stream<T> filter(Predicate<? super T> p)`**
Возвращает поток данных, содержащий все его элементы, совпадающие с указанным предикатом `p`.
- **`long count()`**
Возвращает количество элементов в исходном потоке данных. Это окончая операция.
- **`default Stream<E> stream()`**
- **`default Stream<E> parallelStream()`**
Возвращают последовательный или параллельный поток данных, состоящий из элементов исходной коллекции.

1.2. Создание потока данных

Как было показано выше, любую коллекцию можно преобразовать в поток данных методом `stream()` из интерфейса `Collection`. Если же вместо коллекции имеется массив, то для этой цели служит метод `Stream.of()`:

```
Stream<String> words = Stream.of(contents.split("\\PL+"));
// Метод split() возвращает массив типа String[]
```

У метода `of()` имеются аргументы переменной длины, и поэтому поток данных можно создать из любого количества аргументов, как показано ниже. А для создания потока данных из части массива служит метод `Arrays.stream(array, from, to)`.

```
Stream<String> song = Stream.of("gently", "down", "the", "stream");
```

Чтобы создать поток данных без элементов, достаточно вызвать статический метод `Stream.empty()` следующим образом:

```
Stream<String> silence = Stream.empty();
// Обобщенный тип <String> выводится автоматически;
// что равнозначно вызову Stream.<String>empty()
```

Для создания бесконечных потоков данных в интерфейсе `Stream` имеются два статических метода. В частности, метод `generate()` принимает функцию без аргументов (а формально — объект функционального интерфейса `Supplier<T>`). Всякий раз, когда требуется потоковое значение, эта функция вызывается для получения данного значения. Например, поток постоянных значений можно получить так:

```
Stream<String> echos = Stream.generate(() -> "Echo");
```

а поток случайных чисел следующим образом:

```
Stream<Double> randoms = Stream.generate(Math::random);
```

Для получения бесконечных последовательностей вроде 0 1 2 3 ... служит метод `iterate()`. Он принимает начальное значение и функцию (а формально — объект функционального интерфейса `UnaryOperator<T>`) и повторно применяет функцию к предыдущему результату, как показано в следующем примере кода:

```
Stream<BigInteger> integers =
    Stream.iterate(BigInteger.ZERO, n -> n.add(BigInteger.ONE));
```

Первым элементом такой последовательности является начальное значение `BigInteger.ZERO`; вторым элементом — значение, получаемое в результате вызова функции `f(seed)`, или 1 (как крупное целочисленное значение); следующим элементом — значение, получаемое в результате вызова функции `f(f(seed))`, или 2 и т.д., где `seed` — начальное значение.



НА ЗАМЕТКУ! В прикладном программном интерфейсе Java API имеется целый ряд методов, возвращающих потоки данных. Так, в классе `Pattern` имеется метод `splitAsStream()`, разделяющий последовательность символов типа `CharSequence` по регулярному выражению. Например, для разделения символьной строки на отдельные слова можно воспользоваться следующим оператором:

```
Stream<String> words = Pattern.compile("\\PL+").
    splitAsStream(contents);
```

А статический метод `Files.lines()` возвращает поток данных типа `Stream`, содержащий все строки из файла, как показано ниже.

```
try (Stream<String> lines = Files.lines(path)) {
    Обработать строки
}
```

В примере кода из листинга 1.2 демонстрируются различные способы создания потока данных.

Листинг 1.2. Исходный код из файла streams/CreatingStreams.java

```
1 package streams;
2
3 import java.io.IOException;
4 import java.math.BigInteger;
5 import java.nio.charset.StandardCharsets;
6 import java.nio.file.Files;
7 import java.nio.file.Path;
8 import java.nio.file.Paths;
9 import java.util.List;
10 import java.util.regex.Pattern;
11 import java.util.stream.Collectors;
12 import java.util.stream.Stream;
13
14 public class CreatingStreams
15 {
16     public static <T> void show(String title, Stream<T> stream)
17     {
18         final int SIZE = 10;
19         List<T> firstElements = stream
20             .limit(SIZE + 1)
21             .collect(Collectors.toList());
22         System.out.print(title + ": ");
23         for (int i = 0; i < firstElements.size(); i++)
24         {
25             if (i > 0) System.out.print(", ");
26             if (i < SIZE) System.out.print(firstElements.get(i));
27             else System.out.print("...");
28         }
29         System.out.println();
30     }
31
32     public static void main(String[] args) throws IOException
33     {
34         Path path = Paths.get("../gutenberg/alice30.txt");
35         String contents = new String(Files.readAllBytes(path),
36             StandardCharsets.UTF_8);
37
38         Stream<String> words = Stream.of(contents.split("\\PL+"));
39         show("words", words);
40         Stream<String> song =
41             Stream.of("gently", "down", "the", "stream");
42         show("song", song);
43         Stream<String> silence = Stream.empty();
44         show("silence", silence);
45
46         Stream<String> echos = Stream.generate(() -> "Echo");
47         show("echos", echos);
48
49         Stream<Double> randoms = Stream.generate(Math::random);
50         show("randoms", randoms);
51
52         Stream<BigInteger> integers = Stream.iterate(BigInteger.ONE,
53             n -> n.add(BigInteger.ONE));
54         show("integers", integers);
55
56         Stream<String> wordsAnotherWay =
57             Pattern.compile("\\PL+").splitAsStream(56 contents);
58         show("wordsAnotherWay", wordsAnotherWay);
```

```
59
60     try (Stream<String> lines =
61         Files.lines(path, StandardCharsets.UTF_8))
62     {
63         show("lines", lines);
64     }
65 }
66 }
```

java.util.stream.Stream 8

- **static <T> Stream<T> of(T... values)**

Возвращает поток данных, элементами которого являются заданные значения.

- **static <T> Stream<T> empty()**

Возвращает поток данных без элементов.

- **static <T> Stream<T> generate(Supplier<T> s)**

Возвращает бесконечный поток данных, элементы которого составляются путем повторного вызова функции *s()*.

- **static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)**

Возвращает бесконечный поток данных, элементы которого содержат начальные значения *seed*, функция *f()* сначала вызывается с начальным значением *seed*, а затем со значением предыдущего элемента и т.д.

java.util.Arrays 1.2

- **static <T> Stream<T> stream(T[] array,
int startInclusive, int endExclusive) 8**

Возвращает поток данных, элементы которого сформированы из заданного диапазона в указанном массиве.

java.util.regex.Pattern

- **Stream<String> splitAsStream(CharSequence input) 8**

Возвращает поток данных, элементы которого являются частями последовательности символов *input*, разделяемых по данному шаблону.

java.nio.file.Files 7

- **static Stream<String> lines(Path path) 8**

- **static Stream<String> lines(Path path, Charset cs) 8**

Возвращают поток данных, элементы которого составляют строки из указанного файла в кодировке UTF-8 или в заданном наборе символов.

```
java.util.function.Supplier<T> 8
```

- **T get()**
Возвращает получаемое значение.

1.3. Методы filter(), map() и flatMap()

В результате преобразования потока данных получается другой поток данных, элементы которого являются производными от элементов исходного потока. Ранее демонстрировалось преобразование методом `filter()`, в результате которого получается новый поток данных с элементами, удовлетворяющими определенному условию. А в приведенном ниже примере кода поток символьных строк преобразуется в другой поток, содержащий только длинные слова. В качестве аргумента метода `filter()` указывается объект типа `Predicate<T>`, т.е. функция, преобразующая тип `T` в логический тип `boolean`.

```
List<String> words = ...;
Stream<String> longWords = words.stream().filter(w -> w.length() > 12);
```

Нередко требуется каким-то образом преобразовать значения в потоке данных. Для этой цели можно воспользоваться методом `map()`, передав ему функцию, которая и выполняет нужное преобразование. Например, буквы во всех словах можно сделать строчными следующим образом:

```
Stream<String> lowercaseWords = words.stream().map(String::toLowerCase);
```

В данном примере методу `map()` была передана ссылка на метод. Но вместо нее нередко передается лямбда-выражение, как показано ниже. Получающийся в итоге поток данных содержит первую букву каждого слова.

```
Stream<String> firstLetters = words.stream().map(s -> s.substring(0, 1));
```

При вызове метода `map()` передаваемая ему функция применяется к каждому элементу потока данных, в результате чего образуется новый поток данных с полученными результатами. А теперь допустим, что имеется метод, возвращающий не одно значение, а поток значений, как показано ниже. Например, в результате вызова `letters("boat")` образуется поток данных `["b", "o", "a", "t"]`.

```
public static Stream<String> letters(String s) {
    List<String> result = new ArrayList<>();
    for (int i = 0; i < s.length(); i++)
        result.add(s.substring(i, i + 1));
    return result.stream();
}
```



НА ЗАМЕТКУ! Приведенный выше метод `letters()` можно реализовать намного изящнее с помощью метода `IntStream.range()`, рассматриваемого далее, в разделе 1.13.

Допустим, метод `letters()` передается методу `map()` для преобразования потока символьных строк следующим образом:

```
Stream<Stream<String>> result = words.stream().map(w -> letters(w));
```

В итоге получится поток потоков вроде [... ["у", "о", "у", "р"], ["б", "о", "а", "т"], ...]. Чтобы свести его к потоку букв [... "у", "о", "у", "р", "б", "о", "а", "т", ...], вместо метода `map()` следует вызвать метод `flatMap()` таким образом:

```
Stream<String> flatResult = words.stream().flatMap(w -> letters(w))
// Вызывает метод letters() для каждого слова и сводит результаты
```

 **НА ЗАМЕТКУ!** Аналогичный метод `flatMap()` можно обнаружить и в других классах, а не в тех, что представляют потоки данных. Это общий принцип вычислительной техники. Допустим, имеется обобщенный тип **G** (например, `Stream`) и функции **f()** и **g()**, преобразующие некоторый тип **T** в тип **G<U>**, а тип **U** — в тип **G<V>** соответственно. В таком случае эти функции можно составить вместе, используя метод `flatMap()`, т.е. применить сначала функцию **f()**, а затем функцию **g()**. В этом состоит главная идея теории монад. Впрочем, метод `flatMap()` можно применять, и не зная ничего о монадах.

java.util.stream.Stream 8

- `Stream<T> filter(Predicate<? super T> predicate)`
Возвращает поток данных, элементы которого совпадают с указанным предикатом.
- `<R> Stream<R> map(Function<? super T, ? extends R> mapper)`
Возвращает поток данных, содержащий результаты применения функции `mapper()` к элементам исходного потока данных.
- `<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)`
Возвращает поток данных, получаемый сцеплением результатов применения функции `mapper()` к элементам исходного потока данных. (Следует иметь в виду, что каждый результат представляет собой отдельный поток данных.)

1.4. Извлечение подпотоков и сцепление потоков данных

В результате вызова `поток.limit(n)` возвращается поток данных, оканчивающийся после `n` элементов или по завершении исходного потока данных, если тот оказывается более коротким. Метод `limit()` особенно удобен для ограничения бесконечных потоков данных до определенной длины. Так, в следующей строке кода получается поток данных, состоящий из 100 произвольных чисел:

```
Stream<Double> randoms = Stream.generate(Math::random).limit(100);
```

А в результате вызова `поток.skip(n)` происходит совершенно противоположное: отбрасываются первые `n` элементов. Если вернуться к рассмотренному ранее примеру чтения текста книги, то в силу особенностей работы метода `split()` первым элементом потока данных оказывается нежелательная пустая строка. От нее можно избавиться, вызвав метод `skip()` следующим образом:

```
Stream<String> words = Stream.of(contents.split("\\PL+")).skip(1);
```

Два потока данных можно соединить вместе с помощью статического метода `concat()` из интерфейса `Stream`, как показано ниже. Разумеется, первый из этих

потоков не должен быть бесконечным, иначе второй поток вообще не сможет соединиться с ним.

```
Stream<String> combined = Stream.concat(
    letters("Hello"), letters("World"));
// В итоге получается следующий поток данных:
// ["H", "e", "l", "l", "o", "W", "o", "r", "l", "d"]
```

java.util.stream.Stream 8

- **Stream<T> limit(long maxSize)**

Возвращает поток данных, состоящий из элементов исходного потока данных вплоть до заданной длины **maxSize**.

- **Stream<T> skip(long n)**

Возвращает поток данных, все элементы которого, кроме начальных **n** элементов, взяты из исходного потока данных.

- **static <T> Stream<T> concat(Stream<? extends T> a,
 Stream<? extends T> b)**

Возвращает поток данных, элементы которого последовательно составлены из элементов потока **a** и элементов потока **b**.

1.5. Другие операции преобразования потоков данных

Метод `distinct()` возвращает поток данных, получающий свои элементы из исходного потока данных в том же самом порядке, за исключением того, что дубликаты в нем подавляются. Очевидно, что во избежание дубликатов в потоке должны запоминаться обнаруженные ранее элементы, как показано ниже.

```
Stream<String> uniqueWords =
    Stream.of("merrily", "merrily", "merrily", "gently").distinct();
// В итоге возвращается только одна строка "merrily"
```

Для сортировки потоков данных имеется несколько вариантов метода `sorted()`. Один из них служит для обработки потоков данных, состоящих из элементов типа `Comparable`, а другой принимает в качестве параметра компаратор типа `Comparator`. В следующем примере кода символьные строки сортируются таким образом, чтобы первой в потоке данных следовала самая длинная строка:

```
Stream<String> longestFirst =
    words.stream().sorted(Comparator.comparing(String::length).reversed());
```

Как и во всех остальных операциях преобразования потоков данных, метод `sorted()` выдает новый поток данных, элементы которого берутся из исходного потока и располагаются в отсортированном порядке. Разумеется, коллекцию можно отсортировать, не прибегая к потокам данных. Метод `sorted()` удобно применять в том случае, если процесс сортировки является частью поточного конвейера.

И наконец, метод `peek()` выдает другой поток данных с теми же самыми элементами, что и у исходного потока, но передаваемая ему функция вызывается всякий раз, когда извлекается элемент. Это удобно для целей отладки, как показано ниже.

```
Object[] powers = Stream.iterate(1.0, p -> p * 2)
    .peek(e -> System.out.println("Fetching " + e))
    .limit(20).toArray();
```

Сообщение выводится в тот момент, когда элемент доступен в потоке данных. Подобным образом можно проверить, что бесконечный поток данных, возвращаемый методом `iterate()`, обрабатывается по требованию. Для целей отладки в методе `peek()` можно вызвать метод, в котором устанавливается точка прерывания.

`java.util.stream.Stream` 8

- **`Stream<T> distinct()`**

Возвращает поток данных, состоящий из неповторяющихся элементов исходного потока.

- **`Stream<T> sorted()`**

- **`Stream<T> sorted(Comparator<? super T> comparator)`**

Возвращают поток данных, состоящий из отсортированных элементов исходного потока. Первый метод требует, чтобы элементы были экземплярами класса, реализующего интерфейс `Comparable`.

- **`Stream<T> peek(Consumer<? super T> action)`**

Возвращает поток данных, состоящий из тех же самых элементов, что и у исходного потока, передавая каждый элемент указанной функции `action()` по мере употребления этого элемента.

1.6. Простые методы сведения

А теперь, когда было показано, каким образом осуществляется создание и преобразование потоков данных, мы наконец-то добрались до самого главного — получения ответов на запросы данных из потоков. В этом разделе рассматриваются так называемые *методы сведения*. Они выполняют окончные операции, сводя поток данных к непотоковому значению, которое может быть далее использовано в программе. Ранее уже демонстрировался простой метод сведения `count()`, возвращающий количество элементов в потоке данных.

К числу других простых методов сведения относятся методы `max()` и `min()`, возвращающие наибольшее и наименьшее значения соответственно. Но не все так просто, поскольку эти методы на самом деле возвращают значение типа `Optional<T>`, которое заключает в себе ответ на запрос данных из потока или обозначает, что запрашиваемые данные отсутствуют, поскольку поток оказался пустым. Раньше в подобных случаях возвращалось пустое значение `null`. Но это могло привести к исключениям в связи с пустыми указателями в не полностью протестированной программе. Пользоваться типом `Optional` удобнее для обозначения отсутствующего возвращаемого значения. Более подробно тип `Optional` рассматривается в следующем разделе. Ниже показано, как получить максимальное значение из потока данных.

```
Optional<String> largest = words.max(String::compareToIgnoreCase);
System.out.println("largest: " + largest.getorElse(""));
```

Метод `findFirst()` возвращает первое значение из непустой коллекции. Зачастую он применяется вместе с методом `filter()`. Так, в следующем примере кода обнаруживается первое слово, начинающееся с буквы Q:

```
Optional<String> startsWithQ =
    words.filter(s -> s.startsWith("Q")).findFirst();
```

Если же требуется любое совпадение, а не только первое, то следует воспользоваться методом `findAny()`, как показано ниже. Это оказывается эффективным при распараллеливании потока данных, поскольку поток может известить о любом обнаруженном в нем совпадении, вместо того чтобы ограничиваться только первым совпадением.

```
Optional<String> startsWithQ =
    words.parallel().filter(s -> s.startsWith("Q")).findAny();
```

Если же требуется лишь выяснить, имеется ли вообще совпадение, то следует воспользоваться методом `anyMatch()`, как показано ниже. Этот метод принимает предикатный аргумент, и поэтому ему не требуется метод `filter()`.

```
boolean aWordStartsWithQ =
    words.parallel().anyMatch(s -> s.startsWith("Q"));
```

Имеются также методы `allMatch()` и `noneMatch()`, возвращающие логическое значение `true`, если с предикатом совпадают все элементы в потоке данных или не совпадает ни один из его элементов соответственно. Эти методы также выгодно выполнять в параллельном режиме.

java.util.stream.Stream 8

- `Optional<T> max(Comparator<? super T> comparator)`
- `Optional<T> min(Comparator<? super T> comparator)`

Возвращают максимальный или минимальный элемент из исходного потока данных, используя порядок расположения, который определяет заданный `comparator`, или же пустое значение типа `Optional`, если исходный поток данных пуст. Это окончательные операции.

- `Optional<T> findFirst()`
- `Optional<T> findAny()`

Возвращают первый или любой элемент из исходного потока данных или же значение типа `Optional`, если исходный поток данных пуст. Это окончательные операции.

- `boolean anyMatch(Predicate<? super T> predicate)`
- `boolean allMatch(Predicate<? super T> predicate)`
- `boolean noneMatch(Predicate<? super T> predicate)`

Возвращают логическое значение `true`, если с заданным предикатом совпадают любые или все элементы исходного потока данных или же не совпадает ни один из его элементов.

1.7. Тип `Optional`

Объект типа `Optional<T>` служит оболочкой для объекта обобщенного типа `T` или же для одного из объектов. В первом случае считается, что значение

присутствует. Тип `Optional<T>` служит в качестве более надежной альтернативы ссылке на обобщенный тип `T`, которая делается на объект или оказывается пустой. Но этот тип надежнее, если правильно им пользоваться. В следующем разделе поясняется, как это делается.

1.7.1. Как обращаться с необязательными значениями

Для эффективного применения типа `Optional` самое главное — выбрать метод, который возвращает *альтернативный вариант*, если значение отсутствует, или *употребляет значение*, если только оно присутствует. Рассмотрим первую методику обращения с необязательными значениями. Нередко имеется значение, возможно, пустая строка "", которое требуется использовать по умолчанию в отсутствие совпадения:

```
String result = optionalString.orElse("");
// Заключенная в оболочку строка,
// а в ее отсутствие - пустая строка ""
```

Кроме того, можно вызвать функцию для вычисления значения по умолчанию следующим образом:

```
String result = optionalString.orElseGet(() ->
    System.getProperty("user.dir"));
// Функция вызывается только по мере необходимости
```

А с другой стороны, в отсутствие значения можно сгенерировать исключение таким образом:

```
String result =
    optionalString.orElseThrow(IllegalStateException::new);
// Предоставить метод, возвращающий объект исключения
```

В приведенных выше примерах было показано, как получить альтернативный вариант, если значение отсутствует. Другая методика обращения с необязательными значениями состоит в том, чтобы употребить значение, если только оно присутствует.

Метод `ifPresent()` принимает функцию в качестве аргумента, как показано ниже. Если необязательное значение существует, оно передается данной функции. В противном случае ничего не происходит.

```
optionalValue.ifPresent(v -> Обработать v);
```

Так, если требуется ввести значение во множество, при условии, что оно существует, достаточно сделать следующий вызов:

```
optionalValue.ifPresent(v -> results.add(v));
```

или просто

```
optionalValue.ifPresent(results::add);
```

В результате вызова метода `ifPresent()` передаваемая ему функция никакого значения не возвращает. Если же требуется обработать результат выполнения функции, следует вызвать метод `map()`, как показано ниже. В итоге переменная `added` будет содержать одно из следующих трех значений: логическое значение `true` или `false`, заключенное в оболочку типа `Optional`, если необязательно

значение `optionalValue` присутствовало, а иначе — пустое значение типа `Optional`.

```
Optional<Boolean> added = optionalValue.map(results::add);
```



НА ЗАМЕТКУ! Метод `map()` служит аналогом метода `map()` из интерфейса `Stream`, упомянутого в разделе 1.3. Необязательное значение можно рассматривать в качестве потока данных нулевой или единичной длины. Получаемый в итоге результат также имеет нулевую или единичную длину. И в последнем случае применялась функция.

java.util.Optional 8

- `TorElse(T other)`

Возвращает присущее значение типа `Optional` или другое значение `other`, если присущее значение типа `Optional` оказывается пустым.

- `T orElseGet(Supplier<? extends T> other)`

Возвращает присущее значение типа `Optional` или результат вызова функции `other()`, если присущее значение типа `Optional` оказывается пустым.

- `<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)`

Возвращает присущее значение типа `Optional` или результат вызова функции `exceptionSupplier()`, если присущее значение типа `Optional` оказывается пустым.

- `void ifPresent(Consumer<? super T> consumer)`

Передает присущее значение типа `Optional` функции `consumer()`, если это значение оказывается непустым.

- `<U> Optional<U> map(Function<? super T, ? extends U> mapper)`

Возвращает результат передачи присущего значения типа `Optional` функции `mapper()`, если это значение оказывается непустым и результат не равен `null`, а иначе — пустое значение типа `Optional`.

1.7.2. Как не следует обращаться с необязательными значениями

Если необязательные значения типа `Optional` не применяются правильно, то они не дают никаких преимуществ по сравнению с прежним подходом, предоставившим выбор между чем-то существующим или несуществующим, т.е. `null`. Метод `get()` получает заключенный в оболочку элемент значения типа `Optional`, если это значение существует, а иначе — он генерирует исключение типа `NoSuchElementException`. Таким образом, следующий фрагмент кода:

```
Optional<T> optionalValue = ...;
optionalValue.get().someMethod()
```

не надежнее, чем такой код:

```
T value = ...;
value.someMethod();
```

Метод `isPresent()` извещает, содержит ли значение объект типа `Optional<T>`. Но выражение

```
if (optionalValue.isPresent()) optionalValue.get().someMethod();
не проще, чем выражение
if (value != null) value.someMethod();
```

java.util.Optional 8

- **T get()**

Возвращает присутствующее значение типа **Optional** или генерирует исключение типа **NoSuchElementException**, если это значение оказывается пустым.

- **boolean isPresent()**

Возвращает логическое значение **true**, если присутствующее значение типа **Optional** оказывается непустым.

1.7.3. Формирование необязательных значений

До сих пор обсуждалось, как употреблять объект типа **Optional**, созданный кем-то другим. Если же требуется написать метод, создающий объект типа **Optional**, то для этой цели имеется несколько статических методов. В приведенном ниже примере демонстрируется применение двух таких методов: **Optional.of(result)** и **Optional.empty()**.

```
public static Optional<Double> inverse(Double x) {
    return x == 0 ? Optional.empty() : Optional.of(1 / x);
}
```

Метод **ofNullable()** служит в качестве моста между возможными пустыми (**null**) и необязательными (**Optional**) значениями. Так, при вызове метода **Optional.ofNullable(obj)** возвращается результат вызова метода **Optional.of(obj)**, если объект **obj** не является пустым (**null**), а иначе — результат вызова метода **Optional.empty()**.

java.util.Optional 8

- **static <T> Optional<T> of(T value)**

- **static <T> Optional<T> ofNullable(T value)**

Возвращают результат типа **Optional** с заданным значением. Если заданное значение **value** равно **null**, то первый метод генерирует исключение типа **NullPointerException**, а второй метод возвращает пустое значение типа **Optional**.

- **static <T> Optional<T> empty()**

Возвращает пустое значение типа **Optional**.

1.7.4. Сочетание функций необязательных значений с методом **flatMap()**

Допустим, имеется метод **f()**, возвращающий объект типа **Optional<T>**, а у целевого типа **T** — метод **g()**, возвращающий объект типа **Optional<U>**. Если бы это были обычные методы, их можно было бы составить в вызов **s.f().g()**. Но

такое сочетание не годится, поскольку результат вызова `s.f()` относится к типу `Optional<T>`, а не к типу `T`. Вместо этого нужно сделать следующий вызов:

```
Optional<U> result = s.f().flatMap(T::g);
```

Если объект, получаемый в результате вызова `s.f()`, присутствует, то к нему применяется метод `g()`. В противном случае возвращается пустой объект типа `Optional<U>`.

Очевидно, что данный процесс можно повторить, если имеются другие методы или лямбда-выражения, возвращающие необязательные значения типа `Optional`. В таком случае из них можно составить конвейер, связав их вызовы в цепочку с методом `flatMap()`, который будет успешно завершен, если завершатся все остальные части конвейера.

В качестве примера рассмотрим надежный метод `inverse()` из предыдущего раздела. Допустим, имеется также следующий надежный метод для извлечения квадратного корня:

```
public static Optional<Double> squareRoot(Double x) {
    return x < 0 ? Optional.empty() : Optional.of(Math.sqrt(x));
}
```

В таком случае извлечь квадратный корень из значения, возвращаемого методом `inverse()`, можно следующим образом:

```
Optional<Double> result = inverse(x).flatMap(MyMath::squareRoot);
```

или таким способом, если он предпочтительнее:

```
Optional<Double> result =
    Optional.of(-4.0).flatMap(Demo::inverse).flatMap(Demo::squareRoot);
```

Если метод `inverse()` или `squareRoot()` возвратит результат вызова метода `Optional.empty()`, то конечный результат окажется пустым.



НА ЗАМЕТКУ! Как было показано в разделе 1.3, метод `flatMap()` из интерфейса `Stream` служит для составления двух других методов, получающих потоки данных, сводя их в результирующий поток потоков. Аналогичным образом действует и метод `Optional.flatMap()`, если необязательное значение интерпретируется как поток данных нулевой или единичной длины.

В примере кода из листинга 1.3 демонстрируется прикладной программный интерфейс API для необязательного типа `Optional`.

Листинг 1.3. Исходный код из файла optional/OptionalTest.java

```
1 package optional;
2
3 import java.io.*;
4 import java.nio.charset.*;
5 import java.nio.file.*;
6 import java.util.*;
7
8 public class OptionalTest
9 {
10    public static void main(String[] args) throws IOException
```

```
11  {
12      String contents = new String(Files.readAllBytes(
13          Paths.get("../gutenberg/alice30.txt")),
14          StandardCharsets.UTF_8);
15      List<String> wordList =
16          Arrays.asList(contents.split("\\PL+"));
17
18      Optional<String> optionalValue = wordList.stream()
19          .filter(s -> s.contains("fred"))
20          .findFirst();
21      System.out.println(optionalValue.orElse("No word")
22          + " contains fred");
23
24      Optional<String> optionalString = Optional.empty();
25      String result = optionalString.orElse("N/A");
26      System.out.println("result: " + result);
27      result = optionalString.orElseGet(() ->
28          Locale.getDefault().getDisplayName());
29      System.out.println("result: " + result);
30  try
31  {
32      result =
33          optionalString.orElseThrow(IllegalStateException::new);
34      System.out.println("result: " + result);
35  }
36  catch (Throwable t)
37  {
38      t.printStackTrace();
39  }
40
41      optionalValue = wordList.stream()
42          .filter(s -> s.contains("red"))
43          .findFirst();
44      optionalValue.ifPresent(
45          s -> System.out.println(s + " contains red"));
46
47      Set<String> results = new HashSet<>();
48      optionalValue.ifPresent(results::add);
49      Optional<Boolean> added = optionalValue.map(results::add);
50      System.out.println(added);
51
52      System.out.println(inverse(4.0)
53          .flatMap(OptionalTest::squareRoot));
54      System.out.println(inverse(-1.0)
55          .flatMap(OptionalTest::squareRoot));
56      System.out.println(inverse(0.0)
57          .flatMap(OptionalTest::squareRoot));
58      Optional<Double> result2 = Optional.of(-4.0)
59          .flatMap(OptionalTest::inverse)
60          .flatMap(OptionalTest::squareRoot);
61      System.out.println(result2);
62  }
63
64  public static Optional<Double> inverse(Double x)
65  {
66      return x == 0 ? Optional.empty() : Optional.of(1 / x);
67  }
```

```

69  public static Optional<Double> squareRoot(Double x)
70  {
71      return x < 0 ? Optional.empty() : Optional.of(Math.sqrt(x));
72  }
73 }
```

java.util.Optional 8

- <U> **Optional<U> flatMap(Function<? super T, Optional<U>> mapper)**

Возвращает результат применения функции **mapper()** к присутствующему значению типа **Optional** или же пустое значение типа **Optional**, если присутствующее значение типа **Optional** оказывается пустым.

1.8. Накопление результатов

По завершении обработки потока данных нередко требуется просмотреть полученные результаты. С этой целью можно вызвать метод **iterate()**, предоставляющий устаревший итератор, которым можно воспользоваться для обхода элементов. С другой стороны, можно вызвать метод **forEach()**, чтобы применить функцию к каждому элементу следующим образом:

```
stream.forEach(System.out::println);
```

В параллельном потоке данных метод **forEach()** выполняет обход элементов в произвольном порядке. Если же их требуется обработать в потоковом порядке, то следует вызвать метод **forEachOrdered()**. Разумеется, в этом случае могут быть утрачены некоторые или даже все преимущества параллелизма.

Но чаще всего результаты требуется накапливать в структуре данных. С этой целью можно вызвать метод **toArray()** и получить элементы из потока данных.

Создать обобщенный массив во время выполнения невозможно, и поэтому в результате вызова **stream.toArray()** возвращается массив типа **Object[]**. Если же требуется массив нужного типа, этому методу следует передать конструктор такого массива, как показано ниже.

```
String[] result = stream.toArray(String[]::new);
// В результате вызова метода stream.toArray()
// получается массив типа Object[]
```

Для накопления элементов потока данных с другой целью имеется удобный метод **collect()**, принимающий экземпляр класса, реализующего интерфейс **Collector**. В частности, класс **Collectors** предоставляет немало фабричных методов для наиболее употребительных коллекторов. Так, для накопления потока данных в списке или множестве достаточно сделать один из следующих вызовов:

```
List<String> result = stream.collect(Collectors.toList());
```

или

```
Set<String> result = stream.collect(Collectors.toSet());
```

Если же требуется конкретная разновидность получаемого множества, то нужно сделать следующий вызов:

```
TreeSet<String> result =
    stream.collect(Collectors.toCollection(TreeSet::new));
```

Допустим, требуется накапливать все символьные строки, сцепляя их. С этой целью можно сделать следующий вызов:

```
String result = stream.collect(Collectors.joining());
```

А если требуется разделитель элементов, то его можно передать методу joining() следующим образом:

```
String result = stream.collect(Collectors.joining(", "));
```

И если поток данных содержит объекты, отличающиеся от символьных строк, их нужно сначала преобразовать в символьные строки:

```
String result =
    stream.map(Object::toString).collect(Collectors.joining(", "));
```

Если результаты обработки потока данных требуется свести к сумме, среднему, максимуму или минимуму, воспользуйтесь методами типа summarizing(Int|Long|Double). Эти методы принимают функцию, преобразующую потоковые объекты в число и возвращающую результат типа (Int|Long|Double) SummaryStatistics, одновременно вычисляя сумму, среднее, максимум и минимум, как показано ниже.

```
IntSummaryStatistics summary = stream.collect(
    Collectors.summarizingInt(String::length));
double averageWordLength = summary.getAverage();
double maxWordLength = summary.getMax();
```

java.util.stream.BaseStream 8

- **Iterator<T> iterator()**

Возвращает итератор для получения элементов исходного потока данных. Это окончаяная операция.

В примере кода из листинга 1.4 демонстрируется порядок накопления элементов из потока данных.

Листинг 1.4. Исходный код из файла collecting/CollectingResults.java

```
1 package collecting;
2
3 import java.io.*;
4 import java.nio.charset.*;
5 import java.nio.file.*;
6 import java.util.*;
7 import java.util.stream.*;
8
9 public class CollectingResults
10 {
11     public static Stream<String> noVowels() throws IOException
12     {
13         String contents = new String(Files.readAllBytes(
```

```
14         Paths.get("../gutenberg/alice30.txt")),
15         StandardCharsets.UTF_8);
16     List<String> wordList =
17         Arrays.asList(contents.split("\\PL+"));
18     Stream<String> words = wordList.stream();
19     return words.map(s -> s.replaceAll("[aeiouAEIOU]", ""));
20 }
21
22 public static <T> void show(String label, Set<T> set)
23 {
24     System.out.print(label + ": " + set.getClass().getName());
25     System.out.println("["
26         + set.stream().limit(10).map(Object::toString)
27         .collect(Collectors.joining(", ")) + "]");
28 }
29
30 public static void main(String[] args) throws IOException
31 {
32     Iterator<Integer> iter =
33         Stream.iterate(0, n -> n + 1).limit(10)
34             .iterator();
35     while (iter.hasNext())
36         System.out.println(iter.next());
37
38     Object[] numbers =
39         Stream.iterate(0, n -> n + 1).limit(10).toArray();
40     System.out.println("Object array:" + numbers);
41     // Следует иметь в виду, что это массив типа Object[]
42
43     try
44     {
45         Integer number = (Integer) numbers[0]; // Верно!
46         System.out.println("number: " + number);
47         System.out.println(
48             "The following statement throws an exception:");
49         Integer[] numbers2 = (Integer[]) numbers;
50             // сгенерировать исключение
51     }
52     catch (ClassCastException ex)
53     {
54         System.out.println(ex);
55     }
56
57     Integer[] numbers3 = Stream.iterate(0, n -> n + 1).limit(10)
58         .toArray(Integer[]::new);
59     System.out.println("Integer array: " + numbers3);
60     // Следует иметь в виду, что это массив типа Integer[]
61
62     Set<String> noVowelSet = noVowels()
63         .collect(Collectors.toSet());
64     show("noVowelSet", noVowelSet);
65
66     TreeSet<String> noVowelTreeSet = noVowels().collect(
67         Collectors.toCollection(TreeSet::new));
68     show("noVowelTreeSet", noVowelTreeSet);
69
70     String result = noVowels().limit(10).collect(
71         Collectors.joining());
```

```

72     System.out.println("Joining: " + result);
73     result = noVowels().limit(10)
74         .collect(Collectors.joining(", "));
75     System.out.println("Joining with commas: " + result);
76
77     IntSummaryStatistics summary = noVowels().collect(
78         Collectors.summarizingInt(String::length));
79     double averageWordLength = summary.getAverage();
80     double maxWordLength = summary.getMax();
81     System.out.println(
82         "Average word length: " + averageWordLength);
83     System.out.println("Max word length: " + maxWordLength);
84     System.out.println("forEach:");
85     noVowels().limit(10).forEach(System.out::println);
86 }
87 }
```

java.util.stream.Stream 8

- **void forEach(Consumer<? super T> action)**

Вызывает функцию **action()** для каждого элемента исходного потока данных. Это окончательная операция.

- **Object[] toArray()**

- **<A> A[] toArray(IntFunction<A[]> generator)**

Возвращают массив объектов или объект типа **A**, если им передается ссылка на конструктор **A[]::new**. Это окончательные операции.

- **<R,A> R collect(Collector<? super T, A, R> collector)**

Накапливает элемент в исходном потоке данных, используя заданный коллектор. Для многих коллекторов в классе **Collectors** имеются фабричные методы.

java.util.stream.Collectors 8

- **static <T> Collector<T, ?, List<T>> toList()**

- **static <T> Collector<T, ?, Set<T>> toSet()**

Возвращают коллекторы, накапливающие элементы в списке или множестве.

- **static <T,C extends Collection<T>> Collector<T, ?, C> toCollection(Supplier<C> collectionFactory)**

Возвращает коллектор, накапливающий элементы в произвольной коллекции. Получает ссылку на конструктор объектов коллекции, например **TreeSet::new**.

- **static Collector<CharSequence, ?, String> joining()**

- **static Collector<CharSequence, ?, String> joining(CharSequence delimiter)**

- **static Collector<CharSequence, ?, String> joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)**

Возвращают коллектор, соединяющий символьные строки. Заданный разделитель размещается между строками, а префикс и суффикс — перед первой строкой и после последней строки соответственно. Если разделитель, префикс и суффикс не указаны, их места остаются пустыми.

java.util.stream.Collectors 8 (окончание)

- static <T> Collector<T, ?, IntSummaryStatistics> summarizingInt(ToIntFunction<? super T> mapper)
- static<T> Collector<T, ?, LongSummaryStatistics> summarizingLong(ToLongFunction<? super T> mapper)
- static <T> Collector<T, ?, DoubleSummaryStatistics> summarizingDouble(ToDoubleFunction<? Super T> mapper)

Возвращают коллекторы, производящие объект типа **(Int|Long|Double)SummaryStatistics**, из которого получается подсчет, сумма, среднее, максимум и минимум результатов применения функции **mapper()** к каждому элементу потока данных.

IntSummaryStatistics 8
LongSummaryStatistics 8
DoubleSummaryStatistics 8

- long getCount()
Возвращает подсчет суммированных элементов.
- (int|long|double) getSum()
Возвращают сумму или среднее суммированных элементов или же нуль, если элементы отсутствуют.
- double getAverage()
Возвращают максимум или минимум суммированных элементов или же значение **Integer|Long|Double|.MAX|MIN_VALUE**, если элементы отсутствуют.
- (int|long|double) getMax()
Возвращают максимум или минимум суммированных элементов или же значение **Integer|Long|Double|.MAX|MIN_VALUE**, если элементы отсутствуют.
- (int|long|double) getMin()
Возвращают максимум или минимум суммированных элементов или же значение **Integer|Long|Double|.MAX|MIN_VALUE**, если элементы отсутствуют.

1.9. Накопление результатов в отображениях

Допустим, имеется поток данных типа **Stream<Person>** и его элементы требуется накапливать в отображении, чтобы в дальнейшем искать людей по их идентификационному номеру. Для этой цели служит метод **Collectors.toMap()**, принимающий в качестве двух своих аргументов функции, чтобы получить ключи и значения из отображения, как показано в следующем примере кода:

```
Map<Integer, String> idToName = people.collect(
    Collectors.toMap(Person::getId, Person::getName));
```

В общем случае, когда значения должны быть конкретными элементами, в качестве второго аргумента данному методу предоставляется функция **Function.identity()**:

```
Map<Integer, Person> idToPerson = people.collect(
    Collectors.toMap(Person::getId, Function.identity()));
```

Если же одному и тому же ключу соответствует больше одного элемента, то возникает конфликт и коллектор генерирует исключение типа

`IllegalStateException`. Такое поведение можно изменить, предоставив данному методу в качестве третьего аргумента функцию, разрешающую подобный конфликт и определяющую значение по заданному ключу, исходя из существующего или нового значения. Такая функция может возвратить существующее значение, новое значение или то и другое.

В приведенном ниже примере создается отображение, содержащее региональные настройки для каждого языка в виде ключа, обозначающего название языка в региональных настройках по умолчанию (например, "German"), и значения, обозначающего его локализованное название (например, "Deutsch"). В данном примере не учитывается, что один и тот же язык может встретиться дважды (например, немецкий в Германии и Швейцарии), и поэтому в отображении сохраняется лишь первая запись.

```
Stream<Locale> locales = Stream.of(Locale.getAvailableLocales());
Map<String, String> languageNames = locales.collect(
    Collectors.toMap(
        Locale::getDisplayLanguage,
        Locale::getDisplayLanguage,
        (existingValue, newValue) -> existingValue));
```



НА ЗАМЕТКУ! В этой главе в качестве структуры данных для хранения региональных настроек употребляется класс `Locale`. Подробнее о региональных настройках речь пойдет в главе 7.

А теперь допустим, что требуется выяснить все языки данной страны. Для этой цели понадобится отображение типа `Map<String, Set<String>>`. Например, значением по ключу "Switzerland" является множество [French, German, Italian]. Сначала для каждого языка сохраняется одноэлементное множество. А всякий раз, когда обнаруживается новый язык заданной страны, образуется объединение из существующего и нового множеств:

```
Map<String, Set<String>> countryLanguageSets = locales.collect(
    Collectors.toMap(
        Locale::getDisplayCountry,
        l -> Collections.singleton(l.getDisplayLanguage()),
        (a, b) -> { // объединить множества a и b
            Set<String> union = new HashSet<>(a);
            union.addAll(b);
            return union; }));
```

Более простой способ получения этого отображения будет представлен в следующем разделе. Если же потребуется древовидное отображение типа `TreeMap`, то в качестве четвертого аргумента методу `toMap()` следует предоставить конструктор данного класса. Необходимо также предоставить функцию объединения. Ниже приведен один из примеров из начала этого раздела, переделанный с целью получить отображение типа `TreeMap`.

```
Map<Integer, Person> idToPerson = people.collect(
    Collectors.toMap(
        Person::getId,
        Function.identity(),
        (existingValue, newValue) ->
            { throw new IllegalStateException(); },
        TreeMap::new));
```



НА ЗАМЕТКУ! Каждому из вариантов метода `toMap()` соответствует эквивалентный метод `toConcurrentMap()`, получающий параллельное отображение. Единое параллельное отображение применяется в процессе параллельного накопления. Если же общее отображение применяется вместе с параллельным потоком данных, то такой способ более эффективный, чем объединение множеств. Но в таком случае элементы не накапливаются в потоковом порядке, хотя это обычно не имеет особого значения.

В примере кода из листинга 1.5 демонстрируется накопление потоковых результатов в отображениях.

Листинг 1.5. Исходный код из файла collecting/CollectingIntoMaps.java

```
1 package collecting;
2
3 import java.io.*;
4 import java.util.*;
5 import java.util.function.*;
6 import java.util.stream.*;
7
8 public class CollectingIntoMaps
9 {
10     public static class Person
11     {
12         private int id;
13         private String name;
14
15         public Person(int id, String name)
16         {
17             this.id = id;
18             this.name = name;
19         }
20
21         public int getId()
22         {
23             return id;
24         }
25
26         public String getName()
27         {
28             return name;
29         }
30
31         public String toString()
32         {
33             return getClass().getName() + "[id=" + id + ", "
34                               + "name=" + name + "]";
35         }
36     }
37
38     public static Stream<Person> people()
39     {
40         return Stream.of(new Person(1001, "Peter"),
41                         new Person(1002, "Paul"),
42                         new Person(1003, "Mary"));
43     }
44 }
```

```

45 public static void main(String[] args) throws IOException
46 {
47     Map<Integer, String> idToName = people().collect(
48         Collectors.toMap(Person::getId, Person::getName));
49     System.out.println("idToName: " + idToName);
50
51     Map<Integer, Person> idToPerson = people().collect(
52         Collectors.toMap(Person::getId, Function.identity()));
53     System.out.println("idToPerson: "
54         + idToPerson.getClass().getName() + idToPerson);
55
56     idToPerson = people().collect(
57         Collectors.toMap(Person::getId, Function.identity(), (
58             existingValue, newValue) -> {
59                 throw new IllegalStateException();
60             }, TreeMap::new));
61     System.out.println("idToPerson: "
62         + idToPerson.getClass().getName()
63         + idToPerson);
64
65     Stream<Locale> locales =
66         Stream.of(Locale.getAvailableLocales());
67     Map<String, String> languageNames = locales.collect(
68         Collectors.toMap(
69             Locale::getDisplayLanguage,
70             l -> l.getDisplayLanguage(),
71             (existingValue, newValue) -> existingValue));
72     System.out.println("languageNames: " + languageNames);
73
74     locales = Stream.of(Locale.getAvailableLocales());
75     Map<String, Set<String>> countryLanguageSets =
76     locales.collect(Collectors.toMap(
77         Locale::getDisplayCountry,
78         l -> Collections.singleton(l.getDisplayLanguage()),
79         (a, b) -> { // объединить множества а и б
80             Set<String> union = new HashSet<>(a);
81             union.addAll(b);
82             return union;
83         }));
84     System.out.println(
85         "countryLanguageSets: " + countryLanguageSets);
86 }
87 }
```

java.util.stream.Collectors 8

- static<T,K,U> Collector<T, ?, Map<K, U>> toMap(Function<? super T, ? extends K> keyMapper, Function<? super T, ? extends U> valueMapper)
- static<T, K, U> Collector<T, ?, Map<K, U>> toMap(Function<? super T, ? extends K> keyMapper, Function<? super T, ? extends U> valueMapper, BinaryOperator<U> mergeFunction)
- static <T, K, U, M extends Map<K, U>> Collector<T, ?, M> toMap(Function<? super T, ? extends K> keyMapper, Function<? super T, ? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapSupplier)

java.util.stream.Collectors 8 (окончание)

- static <T, K, U> Collector<T, ?, ConcurrentHashMap<K, U>>
toConcurrentMap(Function<? super T, ? extends K> keyMapper,
Function<? super T, ? extends U> valueMapper)
- static <T, K, U> Collector<T, ?, ConcurrentHashMap<K, U>>
toConcurrentMap(Function<? super T, ? extends K> keyMapper,
Function<? super T, ? extends U> valueMapper, BinaryOperator<U>
mergeFunction)
- static <T, K, U, M extends ConcurrentHashMap<K,U>> Collector<T, ?, M>
toConcurrentMap(Function<? super T, ? extends K> keyMapper,
Function<? super T, ? extends U> valueMapper, BinaryOperator<U>
mergeFunction, Supplier<M> mapSupplier)

Возвращают коллектор, производящий обычное или параллельное отображение. Функции **keyMapper()** и **valueMapper()** применяются к каждому накапливаемому элементу, возвращая запись в виде пары "ключ-значение" из результирующего отображения. По умолчанию генерируется исключение типа **IllegalStateException**, когда два элемента порождают одинаковый ключ. Вместо этого можно применить функцию **mergeFunction()**, объединяющую значения по одному и тому же ключу. По умолчанию получается результирующее отображение типа **HashMap** или **ConcurrentHashMap**. Но вместо этого можно предоставить функцию **mapSupplier()**, возвращающую требующийся экземпляр отображения.

1.10. Группирование и разделение

В предыдущем разделе было показано, как накапливаются все языки заданной страны. Но этот процесс оказался несколько трудоемким, поскольку для каждого значения из отображения пришлось сначала сформировать одноэлементное множество, а затем указать порядок объединения существующего и нового значений. Очень часто из значений с одинаковыми характеристиками образуются группы, и этот процесс непосредственно поддерживается методом **groupingBy()**.

Рассмотрим задачу группирования региональных настроек по странам. Сначала образуется следующее отображение:

```
Map<String, List<Locale>> countryToLocales = locales.collect(  
    Collectors.groupingBy(Locale::getCountry));
```

Функция **Locale::getCountry()** исполняет роль классификатора группирования. Затем все региональные настройки можно отыскать по заданному коду страны, как показано в следующем примере кода:

```
List<Locale> swissLocales = countryToLocales.get("CH");  
// получить региональные настройки [it_CH, de_CH, fr_CH]
```



НА ЗАМЕТКУ! Как известно, все региональные настройки состоят из кода языка (например, код **en** обозначает английский язык) и кода страницы (например, код **US** обозначает Соединенные Штаты). Так, региональные настройки **en_US** описывают английский язык в Соединенных Штатах, а региональные настройки **en_IE** — английский язык в Ирландии. Некоторым странам требуется несколько региональных настроек. Например, региональные настройки **ga_IE**, описывающие гэльский язык в Ирландии в дополнение к упомянутым выше региональным настройкам **en_IE**. А для Швейцарии требуются три региональные настройки, как было показано в предыдущем разделе.

Когда функция классификатора оказывается предикатной (т.е. функцией, возвращающей логическое значение типа `boolean`), элементы потока данных разделяются на основной список с элементами, для которых функция возвращает логическое значение `true`, и дополнительный список. В данном случае эффективнее воспользоваться методом `partitioningBy()`, чем методом `groupingBy()`. Так, в следующем примере кода все региональные настройки разделяются на те, что описывают английский язык, и все остальные:

```
Map<Boolean, List<Locale>> englishAndOtherLocales = locales.collect(
    Collectors.partitioningBy(l -> l.getLanguage().equals("en")));
List<Locale>> englishLocales = englishAndOtherLocales.get(true);
```

 **НА ЗАМЕТКУ!** Если вызвать метод `groupingByConcurrent()`, то в конечном итоге будет получено отображение, которое заполняется параллельно, если оно применяется вместе с параллельным потоком данных. В этом отношении данный метод очень похож на метод `toConcurrentMap()`.

java.util.stream.Collectors 8

- `static<T,K> Collector<T, ?, Map<K, List<T>>>
groupingBy(Function<? super T, ? extends K> classifier)`
- `static <T,K> Collector<T, ?, ConcurrentMap<K, List<T>>>
groupingByConcurrent(Function<? super T, ? extends K> classifier)`
Возвращают коллектор, производящий обычное или параллельное отображение, где ключи являются результатом применения функции `classifier()` ко всем накапливаемым элементам, а значения — списками элементов с одинаковым ключом.
- `static <T> Collector<T,?,Map<Boolean,List<T>>>
partitioningBy(Predicate<? super T> predicate)`
Возвращает коллектор, производящий отображение, где ключи принимают логическое значение `true/false`, а значения являются списками элементов, совпадающих или не совпадающих с заданным предикатом.

1.11. Нисходящие коллекторы

Метод `groupingBy()` формирует множество, значениями которого являются списки. Если требуется каким-то образом обработать эти списки, то следует предоставить **нисходящий коллектор**. Так, если вместо списков требуются множества, можно воспользоваться коллектором `Collectors.toSet()` следующим образом:

```
Map<String, Set<Locale>> countryToLocaleSet = locales.collect(
    groupingBy(Locale::getCountry, toSet()));
```

 **НА ЗАМЕТКУ!** В данном и последующих примерах из этого раздела предполагается статический импорт `java.util.stream.Collectors.*`, чтобы упростить выражения и сделать их более удобочитаемыми.

Для сведения сгруппированных элементов к числам предоставляется ряд следующих коллекторов:

- **counting()** — производит подсчет накопленных элементов. Так, в следующем примере кода подчитывается количество региональных настроек для каждой страны:

```
Map<String, Long> countryToLocaleCounts = locales.collect(
    groupingBy(Locale::getCountry, counting()));
```

- **summing(Int|Long|Double)** — принимает в качестве аргумента функцию, применяет ее к элементам исходящего потока данных и получает их сумму. Так, в следующем примере кода вычисляется суммарное население каждого штата из потока городов:

```
Map<String, Integer> stateToCityPopulation = cities.collect(
    groupingBy(City::getState, summingInt(City::getPopulation)));
```

- **maxBy() и minBy()** — принимают в качестве аргумента компаратор и получают максимальный и минимальный элементы из исходящего потока данных. Так, в следующем примере кода получается самый крупный город в каждом штате:

```
Map<String, City> stateToLargestCity = cities.collect(
    groupingBy(City::getState,
        maxBy(Comparator.comparing(City::getPopulation))));
```

- **mapping()** — применяет функцию к результатам, полученным из исходящего потока данных, но для обработки результатов ему требуется еще один коллектор. Так, в следующем примере кода города группируются по штатам:

```
Map<String, Optional<String>> stateToLongestCityName = cities.collect(
    groupingBy(City::getState,
        mapping(City::getName,
            maxBy(Comparator.comparing(String::length)))));
```

- В каждом штате получаются названия городов, которые сводятся по максимальной длине.

Метод `mapping()` позволяет изящнее решить задачу из предыдущего раздела — собрать все языки, употребляемые в стране. В предыдущем разделе вместо метода `groupingBy()` применялся метод `toMap()`. А в приведенном ниже решении отпадает необходимость объединять отдельные множества.

```
Map<String, Set<String>> countryToLanguages = locales.collect(
    groupingBy(Locale::getDisplayCountry,
        mapping(Locale::getDisplayLanguage,
            toSet())));
```

Если функция группирования или отображения возвращает тип `int`, `long` или `double`, элементы можно накопить в объекте суммарной статистики, как пояснялось в разделе 1.8. Ниже показано, как это делается. А затем из объектов суммарной статистики каждой группы можно получить суммарное, подсчитанное, среднее, минимальное и максимальное значения функции.

```
Map<String, IntSummaryStatistics> stateToCityPopulationSummary =
    cities.collect(
        groupingBy(City::getState,
            summarizingInt(City::getPopulation))));
```



НА ЗАМЕТКУ! Имеются три варианта метода `reducing()`, выполняющие общие операции сведения, описываемые в разделе 1.12.

Коллекторы можно эффективно сочетать вместе, но в итоге получаются весьма запутанные выражения. Поэтому их лучше всего использовать вместе с методом `groupingBy()` или `partitioningBy()` для обработки значений, преобразуемых из нисходящего потока данных. В противном случае непосредственно в потоках данных просто применяются такие методы, как `map()`, `reduce()`, `count()`, `max()` или `min()`.

В примере кода из листинга 1.6 демонстрируется применение нисходящих коллекторов.

Листинг 1.6. Исходный код из файла collecting/DownstreamCollectors.java

```
1 package collecting;
2
3 import static java.util.stream.Collectors.*;
4
5 import java.io.*;
6 import java.nio.file.*;
7 import java.util.*;
8 import java.util.stream.*;
9
10 public class DownstreamCollectors
11 {
12
13     public static class City
14     {
15         private String name;
16         private String state;
17         private int population;
18
19         public City(String name, String state, int population)
20         {
21             this.name = name;
22             this.state = state;
23             this.population = population;
24         }
25
26         public String getName()
27         {
28             return name;
29         }
30
31         public String getState()
32         {
33             return state;
34         }
35
36         public int getPopulation()
37         {
38             return population;
39         }
40     }
41 }
```

```
42 public static Stream<City> readCities(String filename)
43         throws IOException
44 {
45     return Files.lines(Paths.get(filename))
46         .map(l -> l.split(", "))
47         .map(a -> new City(a[0], a[1], Integer.parseInt(a[2])));
48 }
49
50 public static void main(String[] args) throws IOException
51 {
52     Stream<Locale> locales =
53         Stream.of(Locale.getAvailableLocales());
54     locales = Stream.of(Locale.getAvailableLocales());
55     Map<String, Set<Locale>> countryToLocaleSet =
56         locales.collect(groupingBy(
57             Locale::getCountry, toSet()));
58     System.out.println(
59         "countryToLocaleSet: " + countryToLocaleSet);
60
61     locales = Stream.of(Locale.getAvailableLocales());
62     Map<String, Long> countryToLocaleCounts =
63         locales.collect(groupingBy(
64             Locale::getCountry, counting()));
65     System.out.println(
66         "countryToLocaleCounts: " + countryToLocaleCounts);
67
68     Stream<City> cities = readCities("cities.txt");
69     Map<String, Integer> stateToCityPopulation =
70         cities.collect(groupingBy(
71             City::getState,
72             summingInt(City::getPopulation)));
73     System.out.println(
74         "stateToCityPopulation: " + stateToCityPopulation);
75
76     cities = readCities("cities.txt");
77     Map<String, Optional<String>> stateToLongestCityName =
78         cities.collect(groupingBy(
79             City::getState,
80             mapping(City::getName,
81                 maxBy(Comparator.comparing(String::length)))));
82
83     System.out.println(
84         "stateToLongestCityName: " + stateToLongestCityName);
85
86     locales = Stream.of(Locale.getAvailableLocales());
87     Map<String, Set<String>> countryToLanguages =
88         locales.collect(groupingBy(
89             Locale::getDisplayCountry,
90             mapping(Locale::getDisplayLanguage, toSet())));
91     System.out.println(
92         "countryToLanguages: " + countryToLanguages);
93
94     cities = readCities("cities.txt");
95     Map<String, IntSummaryStatistics>
96         stateToCityPopulationSummary =
97             cities.collect(groupingBy(
98                 City::getState,
99                 summarizingInt(City::getPopulation)));
100    System.out.println(stateToCityPopulationSummary.get("NY"));
101
```

```

102     cities = readCities("cities.txt");
103     Map<String, String> stateToCityNames =
104         cities.collect(groupingBy(
105             City::getState,
106             reducing("", City::getName, (s, t) ->
107                 s.length() == 0 ? t : s
108                 + ", " + t)));
109
110     cities = readCities("cities.txt");
111     stateToCityNames = cities.collect(groupingBy(
112         City::getState,
113         mapping(City::getName, joining(", "))));
114     System.out.println("stateToCityNames: " + stateToCityNames);
115 }
116 }
```

java.util.stream.Collectors 8

- static <T> Collector<T, ?, Long> counting()
 Возвращает коллектор, подсчитывающий накапливаемые элементы.
- static <T> Collector<T, ?, Integer> summingInt(ToIntFunction<? super T> mapper)
 Возвращают коллектор, вычисляющий сумму значений, получаемых в результате применения функции `mapper()` к накапливаемым элементам.
- static <T> Collector<T, ?, Long> summingLong(ToLongFunction<? super T> mapper)
 Возвращают коллектор, вычисляющий максимальный или минимальный из накапливаемых элементов, используя порядок расположения, который задает `comparator`.
- static <T> Collector<T, ?, Optional<T>> maxBy(Comparator<? super T> comparator)
 Возвращают коллектор, вычисляющий максимальный или минимальный из накапливаемых элементов, используя порядок расположения, который задает `comparator`.
- static <T> Collector<T, ?, Optional<T>> minBy(Comparator<? super T> comparator)
 Возвращают коллектор, производящий отображение, где ключи являются результатом применения функции `mapper()` к накапливаемым элементам, а значения — результатом накопления элементов по одному и тому же ключу с помощью коллектора `downstream()`.

1.12. Операции сведения

Метод `reduce()` реализует общий механизм для вычисления значения из потока данных. В простейшей форме он принимает двоичную функцию и применяет ее, начиная с первых двух элементов потока данных. Этот механизм проще всего пояснить на следующем примере функции суммирования:

```
List<Integer> values = ...;
Optional<Integer> sum = values.stream().reduce((x, y) -> x + y);
```

В данном примере метод `reduce()` вычисляет сумму $v_0 + v_1 + v_2 + \dots$, где v_i — элементы потока данных. Этот метод возвращает необязательное значение типа `Optional`, поскольку достоверный результат недостижим, если поток данных пуст.



НА ЗАМЕТКУ! В данном примере можно сделать вызов `reduce(Integer::sum)` вместо вызова `reduce((x, y) -> x + y)`.

В общем, если метод `reduce()` выполняет операцию сведения `op`, то она дает результат $v_0 \text{ op } v_1 \text{ op } v_2 \text{ op } \dots$, где $v_i \text{ op } v_{i+1}$ обозначает вызов функции `op(v_i, v_{i+1})`. Эта операция должна быть ассоциативной, т.е. порядок сочетания ее элементов значения не имеет. В математическом обозначении операция $(x \text{ op } y) \text{ op } z$ должна быть равнозначна операции $x \text{ op } (y \text{ op } z)$. Это дает возможность выполнять эффективное сведение в параллельных потоках данных.

Практическую пользу могут принести многие ассоциативные операции, в том числе сложение, умножение, сцепление символьных строк, получение максимума и минимума, объединение и пересечение множеств. Примером операции, которая не является ассоциативной, служит вычитание. Так, $(6 - 3) - 2 \neq 6 - (3 - 2)$.

Нередко имеется *тождественный элемент* `e` вроде `e op x = x`, и он может быть использован в качестве отправной точки для вычисления. Например, `0` является тождественным элементом операции сложения. Ниже приведена вторая форма вызова метода `reduce()`. Тождественный элемент возвращается в том случае, если поток данных пуст и больше не нужно обращаться к классу `Optional`.

```
List<Integer> values = ...;
Integer sum = values.stream().reduce(0, (x, y) -> x + y)
    // Вычисляет результат 0 + v0 + v1 + v2 + ...
```

А теперь допустим, что имеется поток объектов и требуется получить сумму некоторых свойств, например, длину всех символьных строк в потоке. Для этой цели не годится простая форма метода `reduce()`, поскольку в ней требуется функция $(T, T) \rightarrow T$ с одинаковыми типами аргументов и возвращаемого результата. Но в данном случае имеются два разных типа: `String` — для элементов потока данных и `int` — для накапливаемого результата. На этот случай имеется отдельная форма вызова метода `reduce()`.

Прежде всего нужно предоставить функцию накопления $(total, word) \rightarrow total + word.length()$, которая вызывается повторно, образуя сумму нарастающим итогом. Но если вычисление этой суммы распараллелено, то оно разделяется на несколько параллельных вычислений, результаты которых должны быть объединены. Для этой цели предоставляется вторая функция. Ниже приведена полная форма вызова метода `reduce()` в данном случае.

```
int result = words.reduce(0,
    (total, word) -> total + word.length(),
    (total1, total2) -> total1 + total2);
```



НА ЗАМЕТКУ! На практике методом `reduce()` приходится пользоваться нечасто. Ведь на много проще преобразовать исходный поток символьных строк в поток чисел и воспользоваться одним из методов для вычисления суммы, максимума или минимума. [Подробнее потоки чисел рассматриваются далее, в разделе 1.13.] В данном конкретном случае можно было бы сделать вызов `words.mapToInt(String::length).sum()`. Это было бы проще и эффективнее, поскольку не потребовало бы упаковки.



НА ЗАМЕТКУ! Иногда метод `reduce()` оказывается недостаточно обобщенным. Допустим, требуется накопить результаты во множестве типа `BitSet`. Если распараллелить эту коллекцию, то разместить ее элементы в одном множестве типа `BitSet` не удастся, поскольку объект типа `BitSet` не является потокобезопасным. Именно поэтому нельзя воспользоваться методом `reduce()`. Каждый сегмент исходной коллекции должен начинаться со своего пустого множества, а методу `reduce()` можно представить только одно тождественное значение. В таком случае следует воспользоваться методом `collect()`, который принимает следующие аргументы.

Поставщик для получения новых экземпляров целевого объекта. Например, конструктор для построения хеш-множества.

Накопитель, вводящий элемент в целевой объект. Например, метод `add()`.

Объединитель, соединяющий два объекта в один. Например, метод `addAll()`.

Ниже показано, каким образом метод `collect()` вызывается для множества битов.

```
BitSet result = stream.collect(BitSet::new, BitSet::set, BitSet::or);
```

```
java.util.Stream 8
```

- `Optional<T> reduce(BinaryOperator<T> accumulator)`
- `T reduce(T identity, BinaryOperator<T> accumulator)`
- `<U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)`

Формируют накапливаемый итог элементов потока данных с помощью заданной функции `accumulator()`. Если же предоставляется аргумент `identity`, то он становится первым накапливаемым значением. А если предоставляется аргумент `combiner`, то он может быть использован для объединения итогов по сегментам потока данных, которые накапливаются отдельно.

- `<R> R collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner)`

Накапливает элементы в результат типа `R`. Для каждого сегмента потока данных вызывается функция `supplier()`, предостав员ющая первоначальный результат. Функция `accumulator()` вызывается для добавления к нему элементов изменчивым способом, а функция `combiner()` — для объединения обоих результатов.

1.13. Потоки данных примитивных типов

До сих пор целочисленные значения накапливались в потоке данных типа `Stream<Integer>`, несмотря на то, что заключать каждое целочисленное значение в объект-оболочку совершенно неэффективно. Это же относится и к другим примитивным типам данных `double`, `float`, `long`, `short`, `char`, `byte` и `boolean`. В библиотеке потоков данных имеются специализированные классы `IntStream`, `LongStream` и `DoubleStream`, позволяющие сохранять значения примитивных типов непосредственно, не прибегая к помощи оболочек. Так, если требуется сохранить значения типа `short`, `char`, `byte` и `boolean`, достаточно воспользоваться классом `IntStream`, а для хранения значений типа `float` — классом `DoubleStream`.

Чтобы создать поток данных типа `IntStream`, достаточно вызвать методы `IntStream.of()` и `Arrays.stream()` следующим образом:

```
IntStream stream = IntStream.of(1, 1, 2, 3, 5);
stream = Arrays.stream(values, from, to);
// массив values относится к типу int[]
```

К потокам данных примитивных типов, как и к потокам объектов, можно применять статические методы `generate()` и `iterate()`. Кроме того, в классах `IntStream` и `LongStream` имеются статические методы `range()` и `rangeClosed()`, генерирующие диапазоны целочисленных значений с единичным шагом:

```
IntStream zeroToNinetyNine = IntStream.range(0, 100);
// Верхний предел исключительно
IntStream zeroToHundred = IntStream.rangeClosed(0, 100);
// Верхний предел включительно
```

В интерфейсе `CharSequence` имеются методы `codePoints()` и `chars()`, получающие поток типа `IntStream` кодов символов в Юникоде или кодовых единиц в кодировке UTF-16. (Подробнее о кодировках символов см. в главе 2.) Ниже приведен пример применения метода `codePoints()`.

```
String sentence = "\uD835\uDD46 is the set of octonions.";
// \uD835\uDD46 – это кодировка UTF-16 знака ®, обозначающего
// октонионы в Юникоде (U+1D546)
```

```
IntStream codes = sentence.codePoints();
// Поток шестнадцатеричных значений 1D546 20 69 73 20 . . .
```

Поток объектов можно преобразовать в поток данных примитивных типов с помощью метода `mapToInt()`, `mapToLong()` или `mapToDouble()`. Так, если имеется поток символьных строк и их длины требуется обработать как целочисленные значения, это можно сделать и средствами класса `IntStream` следующим образом:

```
Stream<String> words = ....;
IntStream lengths = words.mapToInt(String::length);
```

Чтобы преобразовать поток данных примитивного типа в поток объектов, достаточно воспользоваться методом `boxed()` следующим образом:

```
Stream<Integer> integers = IntStream.range(0, 100).boxed();
```

Как правило, методы для потоков данных примитивных типов аналогичны методам для потоков объектов. Ниже перечислены наиболее существенные их отличия.

- Методы типа `toArray` возвращают массивы примитивных типов.
- Методы, возвращающие результат необязательного типа, возвращают значение типа `OptionalInt`, `OptionalLong` или `OptionalDouble`. Классы этих типов аналогичны классу `Optional`, но у них имеются методы `getAsInt()`, `getAsLong()` и `getAsDouble()` вместо метода `get()`.
- Имеются методы `sum()`, `average()`, `max()` и `min()`, возвращающие сумму, среднее, максимум и минимум соответственно. Эти методы не определены для потоков объектов.
- Метод `summaryStatistics()` возвращает объект типа `IntSummaryStatistics`, `LongSummaryStatistics` или `DoubleSummaryStatistics`, способный одновременно сообщать о сумме, среднем, максимуме и минимуме в потоке данных.



НА ЗАМЕТКУ! В классе `Random` имеются методы `ints()`, `longs()` и `doubles()`, возвращающие потоки данных примитивных типов, состоящие из случайных чисел.

В примере кода из листинга 1.7 демонстрируется применение элементов прикладного программного интерфейса API для потоков данных примитивных типов.

Листинг 1.7. Исходный код из файла streams/PrimitiveTypeStreams.java

```
1 package streams;
2
3 import java.io.IOException;
4 import java.nio.charset.StandardCharsets;
5 import java.nio.file.Files;
6 import java.nio.file.Path;
7 import java.nio.file.Paths;
8 import java.util.stream.Collectors;
9 import java.util.stream.IntStream;
10 import java.util.stream.Stream;
11
12 public class PrimitiveTypeStreams
13 {
14     public static void show(String title, IntStream stream)
15     {
16         final int SIZE = 10;
17         int[] firstElements = stream.limit(SIZE + 1).toArray();
18         System.out.print(title + ": ");
19         for (int i = 0; i < firstElements.length; i++)
20         {
21             if (i > 0) System.out.print(", ");
22             if (i < SIZE) System.out.print(firstElements[i]);
23             else System.out.print("...");
24         }
25         System.out.println();
26     }
27
28     public static void main(String[] args) throws IOException
29     {
30         IntStream is1 = IntStream.generate(() ->
31                                         (int) (Math.random() * 100));
32         show("is1", is1);
33         IntStream is2 = IntStream.range(5, 10);
34         show("is2", is2);
35         IntStream is3 = IntStream.rangeClosed(5, 10);
36         show("is3", is3);
37
38         Path path = Paths.get("../gutenberg/alice30.txt");
39         String contents = new String(Files.readAllBytes(path),
40                                     StandardCharsets.UTF_8);
41
42         Stream<String> words = Stream.of(contents.split("\\PL+"));
```

```

43     IntStream is4 = words.mapToInt(String::length);
44     show("is4", is4);
45     String sentence = "\uD835\uDD46 is the set of octonions.";
46     System.out.println(sentence);
47     IntStream codes = sentence.codePoints();
48     System.out.println(codes.mapToObj(c ->
49                           String.format("%X ", c))
50                           .collect( Collectors.joining())));
51
52     Stream<Integer> integers = IntStream.range(0, 100).boxed();
53     IntStream is5 = integers.mapToInt(Integer::intValue);
54     show("is5", is5);
55 }
56 }
```

java.util.stream.IntStream 8

- **static IntStream range(int *startInclusive*, int *endExclusive*)**
Возвращают поток данных типа **IntStream** с целочисленными элементами в заданном диапазоне.
- **static IntStream rangeClosed(int *startInclusive*, int *endInclusive*)**
Возвращают поток данных типа **IntStream** с целочисленными элементами в заданном диапазоне.
- **static IntStream of(int... *values*)**
Возвращает поток данных типа **IntStream** с заданными элементами.
- **int[] toArray()**
Возвращает массив, состоящий из элементов исходного потока данных.
- **int sum()**
- **OptionalDouble average()**
- **OptionalInt max()**
- **OptionalInt min()**
- **IntSummaryStatistics summaryStatistics()**
Возвращают сумму, среднее, максимум, минимум элементов исходного потока данных или объект, из которого могут быть получены эти четыре результата.
- **Stream<Integer> boxed()**
Возвращает поток объектов-оболочек для элементов исходного потока данных.

java.util.stream.LongStream 8

- **static LongStream range(long *startInclusive*, long *endExclusive*)**
Возвращают поток данных типа **LongStream** с целочисленными элементами в заданном диапазоне.
- **static LongStream rangeClosed(long *startInclusive*, long *endInclusive*)**
Возвращают поток данных типа **LongStream** с целочисленными элементами в заданном диапазоне.
- **static LongStream of(long... *values*)**
Возвращает поток данных типа **LongStream** с заданными элементами.
- **long[] toArray()**
Возвращает массив, состоящий из элементов исходного потока данных.

java.util.stream.LongStream 8 (окончание)

- **long sum()**
- **OptionalDouble average()**
- **OptionalLong max()**
- **OptionalLong min()**
- **LongSummaryStatistics summaryStatistics()**

Возвращают сумму, среднее, максимум, минимум элементов исходного потока данных или объект, из которого могут быть получены эти четыре результата.

- **Stream<Long> boxed()**

Возвращает поток объектов-оболочек для элементов исходного потока данных.

java.util.stream.DoubleStream 8

- **static DoubleStream of(double... values)**

Возвращает поток данных типа **DoubleStream** с заданными элементами.

- **double[] toArray()**

Возвращает массив, состоящий из элементов исходного потока данных.

- **double sum()**

- **OptionalDouble average()**

- **OptionalDouble max()**

- **OptionalDouble min()**

- **DoubleSummaryStatistics summaryStatistics()**

Возвращают сумму, среднее, максимум, минимум элементов исходного потока данных или объект, из которого могут быть получены эти четыре результата.

- **Stream<Double> boxed()**

Возвращает поток объектов-оболочек для элементов исходного потока данных.

java.lang.CharSequence 1.0

- **IntStream codePoints() 8**

Возвращает поток всех кодовых точек исходной символьной строки в Юникоде.

java.util.Random 1.0

- **IntStream ints()**

- **IntStream ints(int randomNumberOrigin, int randomNumberBound) 8**

- **IntStream ints(long streamSize) 8**

- **IntStream ints(long streamSize, int randomNumberOrigin, int randomNumberBound) 8**

java.util.Random 1.0 (окончание)

- **LongStream longs() 8**
- **LongStream longs(long randomNumberOrigin, long randomNumberBound) 8**
- **LongStream longs(long streamSize) 8**
- **LongStream longs(long streamSize, long randomNumberOrigin, long randomNumberBound) 8**
- **DoubleStream doubles() 8**
- **DoubleStream doubles(double randomNumberOrigin, double randomNumberBound) 8**
- **DoubleStream doubles(long streamSize) 8**
- **DoubleStream doubles(long streamSize, double randomNumberOrigin, double randomNumberBound) 8**

Возвращают потоки произвольных чисел. Если указан аргумент **streamSize**, возвращается конечный поток с заданным количеством элементов. Если же предоставлены границы, то возвращается поток с элементами в пределах от **randomNumberOrigin** (включительно) до **randomNumberBound** (исключительно).

java.util.Optional(Int|Long|Double) 8

- **static Optional(Int|Long|Double) of((int|long|double) value)**
Возвращает необязательный объект с предоставленным значением указанного примитивного типа.
- **(int|long|double) getAs(Int|Long|Double)()**
Возвращает значение присутствующего необязательного объекта или генерирует исключение типа **NoSuchElementException**, если этот объект оказывается пустым.
- **(int|long|double) orElse((int|long|double) other)**
- **(int|long|double) orElseGet((Int|Long|Double)Supplier other)**
Возвращают значение присутствующего необязательного объекта или альтернативное значение, если этот объект оказывается пустым.
- **void ifPresent((Int|Long|Double)Consumer consumer)**
Передает значение присутствующего необязательного объекта функции **consumer()**, если этот объект оказывается непустым.

java.util.(Int|Long|Double)SummaryStatistics 8

- **long getCount()**
 - **(int|long|double) getSum()**
 - **double getAverage()**
 - **(int|long|double) getMax()**
 - **(int|long|double) getMin()**
- Возвращают подсчет, сумму, среднее, максимум и минимум накапливаемых элементов исходного потока данных.

1.14. Параллельные потоки данных

Потоки данных упрощают распараллеливание групповых операций. Этот процесс происходит в основном автоматически, но требует соблюдения немногих правил. Прежде всего, нужно иметь в своем распоряжении параллельный поток данных. Получить параллельный поток данных можно из любой коллекции с помощью метода `Collection.parallelStream()`:

```
Stream<String> parallelWords = words.parallelStream();
```

Более того, метод `parallel()` преобразует любой последовательный поток данных в параллельный поток, как показано ниже. При выполнении окончного метода поток данных действует в параллельном режиме, и поэтому промежуточные операции в этом потоке распараллеливаются.

```
Stream<String> parallelWords = Stream.of(wordArray).parallel();
```

Когда потоковые операции выполняются параллельно, то цель состоит в том, чтобы получить в итоге такой же результат, как и в том случае, если бы они выполнялись последовательно. Очень важно, чтобы эти операции можно было выполнять в произвольном порядке.

Допустим, требуется подсчитать все короткие слова в потоке символьных строк. В приведенном ниже примере демонстрируется, как не следует решать эту задачу.

```
int[] shortWords = new int[12];
words.parallelStream().forEach(
    s -> { if (s.length() < 12) shortWords[s.length()]++; });
// ОШИБКА: состояние гонок!
System.out.println(Arrays.toString(shortWords));
```

Приведенный выше код написан очень скверно. Функция, передаваемая методу `forEach()`, выполняется параллельно в нескольких потоках исполнения, в каждом из которых обновляется разделяемый ими общий массив. Как будет показано в главе 14, это классическое *состояние гонок*. Если выполнить данный код многократно, то в результате каждого его выполнения, вероятнее всего, будет получена совсем другая последовательность подсчитанных коротких слов, причем каждый раз неверная.

В обязанности программиста входит обеспечение надежного выполнения в параллельном режиме функций, передаваемых для распараллеливания операций в потоке данных. Для этого лучше всего избегать изменяемого состояния. В следующем примере кода вычисления можно надежно распараллелить, если сгруппировать символьные строки по длине и подсчитать их:

```
Map<Integer, Long> shortWordCounts =
    words.parallelStream()
        .filter(s -> s.length() < 10)
        .collect(groupingBy(
            String::length,
            counting()));
```



Совет! Функции, передаваемые параллельному потоку данных, не должны блокироваться. Для оперирования отдельными сегментами параллельного потока данных применяется пул вилочного соединения. Если же блокируется несколько потоковых операций, то этот пул может оказаться просто неспособным выполнять свои функции.

По умолчанию потоки данных, получаемые из упорядоченных коллекций (массивов и списков), диапазонов, генераторов, итераторов или в результате вызова метода `Stream.sorted()`, упорядочиваются. Результаты накапливаются в порядке следования исходных элементов и полностью предсказуемы. Если выполнить одни и те же операции дважды, то будут получены совершенно одинаковые результаты.

Упорядочение не исключает эффективное распараллеливание. Например, при вызове `stream.map(fun)` поток данных может быть разбит на n сегментов, каждый из которых обрабатывается параллельно. А полученные результаты снова собираются по порядку.

Некоторые операции могут быть распараллелены более эффективно, если требование упорядочения опускается. Вызывая метод `Stream.unordered()`, можно указать, что упорядочение не имеет значения. Это, в частности, выгодно при выполнении операции методом `Stream.distinct()`. В упорядоченном потоке метод `distinct()` сохраняет первый из всех равных элементов. Этим ускоряется распараллеливание, поскольку в потоке исполнения, обрабатывающем отдельный сегмент, неизвестно, какие именно элементы следует отбросить, до тех пор, пока сегмент не будет обработан. Если же допускается сохранить любой однозначный элемент, то все сегменты могут быть обработаны параллельно (с помощью общего множества для отслеживания дубликатов).

Если опустить упорядочение, то можно также ускорить выполнение метода `limit()`. Если же требуется обработать любые n элементов из потока данных и при этом неважно, какие из них будут получены, то с этой целью можно сделать следующий вызов:

```
Stream<String> sample = words.parallelStream().unordered().limit(n);
```

Как обсуждалось в разделе 1.9, объединять отображения невыгодно из-за немалых затрат. Именно поэтому в методе `Collectors.groupingByConcurrent()` используется общее параллельное отображение. Чтобы извлечь выгоду из параллелизма, порядок следования значений в отображении должен быть иным, чем в потоке данных:

```
Map<Integer, List<String>> result = words.parallelStream().collect(
    Collectors.groupingByConcurrent(String::length));
// Значения не накапливаются в потоковом порядке
```

Разумеется, это не имеет особого значения, если применяется исходящий коллектор, не зависящий от упорядочения, как в следующем примере кода:

```
Map<Integer, Long> wordCounts =
    words.parallelStream()
        .collect(
            groupingByConcurrent(
                String::length,
                counting()));
```



Совет! При выполнении потоковой операции очень важно не изменять коллекцию, поддерживающую поток данных, даже если такое изменение и является потокобезопасным. Напомним, что данные в потоках данных не накапливаются, а всегда находятся в отдельной коллекции. Если попытаться изменить коллекцию, то результат выполнения потоковых операций окажется неопределенным. В документации на комплект JDK такое требование называется **невмешательством**. Оно относится как к последовательным, так и к параллельным потокам данных.

Точнее говоря, коллекцию можно изменять вплоть до момента выполнения окончайной операции, поскольку промежуточные потоковые операции выполняются по требованию. Так, следующий фрагмент кода вполне работоспособен, хотя и не рекомендуется:

```
List<String> wordList = ...;
Stream<String> words = wordList.stream();
wordList.add("END");
long n = words.distinct().count();
```

А приведенный ниже фрагмент кода оказывается неработоспособным.

```
Stream<String> words = wordList.stream();
words.forEach(s -> if (s.length() < 12) wordList.remove(s));
// ОШИБКА: вмешательство!
```

Для обеспечения нормальной работы потоков данных необходимо соблюсти ряд следующих условий.

- Данные должны находиться в оперативной памяти. Было бы неэффективно ожидать их поступления из внешнего источника.
- Поток данных должен эффективно разделяться на подобласти. Поток данных, поддерживаемый массивом или сбалансированным двоичным деревом, вполне пригоден, но этого нельзя сказать о результате вызова метода `Stream.iterate()`.
- Потоковые операции должны выполнять значительный объем работы. Если общая рабочая нагрузка невелика, то затраты на организацию параллельных вычислений не оправдываются.
- Потоковые операции не должны блокироваться.

Иными словами, не все потоки данных следует превращать в параллельные. Пользоваться параллельными потоками данных следует лишь в том случае, если постоянно приходится выполнять значительный объем вычислений над данными, уже находящимися в оперативной памяти.

В примере кода из листинга 1.8 показано, как следует обращаться с параллельными потоками данных.

Листинг 1.8. Исходный код из файла parallel/ParallelStreams.java

```
1 package parallel;
2
3 import static java.util.stream.Collectors.*;
4
5 import java.io.*;
6 import java.nio.charset.*;
7 import java.nio.file.*;
```

```
8 import java.util.*;
9 import java.util.stream.*;
10
11 public class ParallelStreams
12 {
13     public static void main(String[] args) throws IOException
14     {
15         String contents = new String(Files.readAllBytes(
16             Paths.get("../gutenberg/alice30.txt")),
17             StandardCharsets.UTF_8);
18         List<String> wordList =
19             Arrays.asList(contents.split("\\PL+"));
20
21         // Далее следует весьма неудачный код
22         int[] shortWords = new int[10];
23         wordList.parallelStream().forEach(s ->
24         {
25             if (s.length() < 10) shortWords[s.length()]++;
26         });
27         System.out.println(Arrays.toString(shortWords));
28
29         // Еще одна попытка, хотя результат, скорее всего,
30         // окажется иным и к тому же неверным
31         Arrays.fill(shortWords, 0);
32         wordList.parallelStream().forEach(s ->
33         {
34             if (s.length() < 10) shortWords[s.length()]++;
35         });
36         System.out.println(Arrays.toString(shortWords));
37
38         // Выход из положения: группирование и подсчет
39         Map<Integer, Long> shortWordCounts = wordList
40             .parallelStream().filter(s -> s.length() < 10)
41             .collect(groupingBy(String::length, counting()));
42
43         System.out.println(shortWordCounts);
44
45         // Несходящий порядок не детерминирован
46         Map<Integer, List<String>> result =
47             wordList.parallelStream().collect(
48                 Collectors.groupingByConcurrent(String::length));
49
50         System.out.println(result.get(14));
51
52         result = wordList.parallelStream().collect(
53             Collectors.groupingByConcurrent(String::length));
54
55         System.out.println(result.get(14));
56
57         Map<Integer, Long> wordCounts =
58             wordList.parallelStream().collect(
59                 groupingByConcurrent(String::length, counting()));
60
61         System.out.println(wordCounts);
62     }
63 }
```

```
java.util.stream.BaseStream<T, S extends BaseStream<T, S>> 8
```

- **S parallel()**

Возвращает параллельный поток данных с такими же элементами, как и у исходного потока.

- **S unordered()**

Возвращает неупорядоченный поток данных с такими же элементами, как и у исходного потока.

```
java.util.Collection<E> 1.2
```

- **Stream<E> parallelStream()** 8

Возвращает параллельный поток данных с элементами из исходной коллекции.

В этой главе было показано, как применять на практике библиотеку потоков данных, внедренную в версии Java 8. В следующей главе рассматривается не менее важная тема организации ввода-вывода.

ГЛАВА

2

Ввод и вывод

В этой главе...

- ▶ Потоки ввода-вывода
- ▶ Ввод-вывод текста
- ▶ Чтение и запись двоичных данных
- ▶ Потоки ввода-вывода и сериализация объектов
- ▶ Манипулирование файлами
- ▶ Файлы, отображаемые в памяти
- ▶ Регулярные выражения

В этой главе речь пойдет о прикладных программных интерфейсах (API), доступных в Java для организации ввода-вывода данных. Из нее вы, в частности, узнаете, как получать доступ к файлам и каталогам, как читать и записывать данные в двоичном и текстовом формате. В главе рассматривается также механизм сериализации, позволяющий так же просто сохранять объекты, как и текстовые или числовые данные. Далее речь пойдет о манипулировании файлами и каталогами. И в завершение главы обсуждаются регулярные выражения, хотя они и не имеют непосредственного отношения к потокам ввода-вывода и файлам. Тем не менее трудно найти более подходящее место для обсуждения этой темы, как его, очевидно, не удалось найти и разработчикам Java, присоединившим спецификацию прикладного программного интерфейса API для регулярных выражений к запросу на уточнение новых средств ввода-вывода.

2.1. Потоки ввода-вывода

В прикладном программном интерфейсе Java API объект, из которого можно читать последовательность байтов, называется *потоком ввода*, а объект, в который

можно записывать последовательность байтов, — *потоком вывода*. В роли таких источников и приемников последовательностей байтов чаще всего выступают файлы, но могут также служить сетевые соединения и даже блоки памяти. Абстрактные классы `InputStream` и `OutputStream` служат основанием для иерархии классов ввода-вывода.



На заметку! Рассматриваемые здесь потоки ввода-вывода никак не связаны с потоками данных, представленными в предыдущей главе. Ради ясности они называются в тексте потоками ввода и вывода всякий раз, когда речь идет о вводе-выводе.

Байтовые потоки ввода-вывода неудобны для обработки информации, хранящейся в Юникоде (напомним, что в Юникоде на каждый символ приходится несколько байтов). Поэтому для обработки символов в Юникоде предусмотрена отдельная иерархия классов, наследующих от абстрактных классов `Reader` и `Writer`. Эти классы позволяют выполнять операции чтения и записи на основании двухбайтовых значений типа `char` (т.е. кодовых единиц в кодировке UTF-16), а не однобайтовых значений типа `byte`.

2.1.1. Чтение и запись байтов

В классе `InputStream` имеется следующий абстрактный метод:

```
abstract int read()
```

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

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

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

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

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

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

```

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

```

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

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

`java.io.InputStream` 1.0

- **abstract int read()**
Считывает байт данных и возвращает его. По достижении конца потока возвращает значение `-1`.
- **int read(byte[] b)**
Считывает данные в байтовый массив и возвращает фактическое количество считанных байтов или значение `-1`, если достигнут конец потока ввода. Этот метод позволяет считать максимум `b.length` байтов.
- **int read(byte[] b, int off, int len)**
Считывает данные в байтовый массив. Возвращает фактическое количество считанных байтов или значение `-1`, если достигнут конец потока ввода.

Параметры:

b

Массив, в который должны
считываться данные

off

Смещение в массиве **b**,
обозначающее позицию, с которой
должно начинаться размещение в нем байтов

len

Максимальное количество считываемых байтов

- **long skip(long n)**

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

java.io.InputStream 1.0 (окончание)

- **int available()**
Возвращает количество байтов, доступных без блокирования. (Напомним, что блокирование означает потерю текущим потоком исполнения своей очереди на выполнение.)
- **void close()**
Закрывает поток ввода.
- **void mark(int readlimit)**
Устанавливает маркер на текущей позиции в потоке ввода. (Не все потоки поддерживают такую функциональную возможность.) Если из потока ввода считано байтов больше заданного предела **readlimit**, в потоке ввода можно пренебречь устанавливаемым маркером.
- **void reset()**
Возвращается к последнему маркеру. Последующие вызовы метода **read()** приводят к повторному считыванию байтов. В отсутствие текущего маркера поток ввода не устанавливается в исходное положение.
- **boolean markSupported()**
Возвращает логическое значение **true**, если в потоке ввода поддерживается возможность устанавливать маркеры.

java.io.OutputStream 1.0

- **abstract void write(int n)**
Записывает байт данных.
- **void write(byte[] b)**
- **void write(byte[] b, int off, int len)**
Записывают все байты или определенный ряд байтов из массива **b**.

Параметры:	b	Массив, из которого должны выбираться данные для записи
	off	Смещение в массиве b , обозначающее позицию, с которой должна начинаться выборка байтов для записи
	len	Общее количество записываемых байтов
- **void close()**
Очищает и закрывает поток вывода.
- **void flush()**
Очищает поток вывода, отправляя любые находящиеся в буфере данные по месту назначения.

2.1.2. Полный комплект потоков ввода-вывода

В отличие от языка С, где для ввода-вывода достаточно и единственного типа FILE*, в Java имеется целый комплект из более чем 60 (!) различных типов потоков

ввода-вывода (рис. 2.1 и 2.2). Разделим эти типы по областям их применения. Так, для классов, обрабатывающих байты и символы, существуют отдельные иерархии.

Как упоминалось выше, классы `InputStream` и `OutputStream` позволяют выполнять чтение и запись отдельных байтов и массивов байтов. Эти классы образуют основу иерархии, приведенной на рис. 2.1. Для чтения и записи символьных строк и чисел требуются подклассы, обладающие немалыми функциональными возможностями. Например, классы `DataInputStream` и `DataOutputStream` позволяют выполнять чтение и запись всех простых типов данных Java в двоичном формате. И, наконец, имеются классы для выполнения отдельных полезных операций ввода-вывода, например классы `ZipInputStream` и `ZipOutputStream`, позволяющие читать и записывать данные в файлы с уплотнением в таком известном формате, как ZIP.

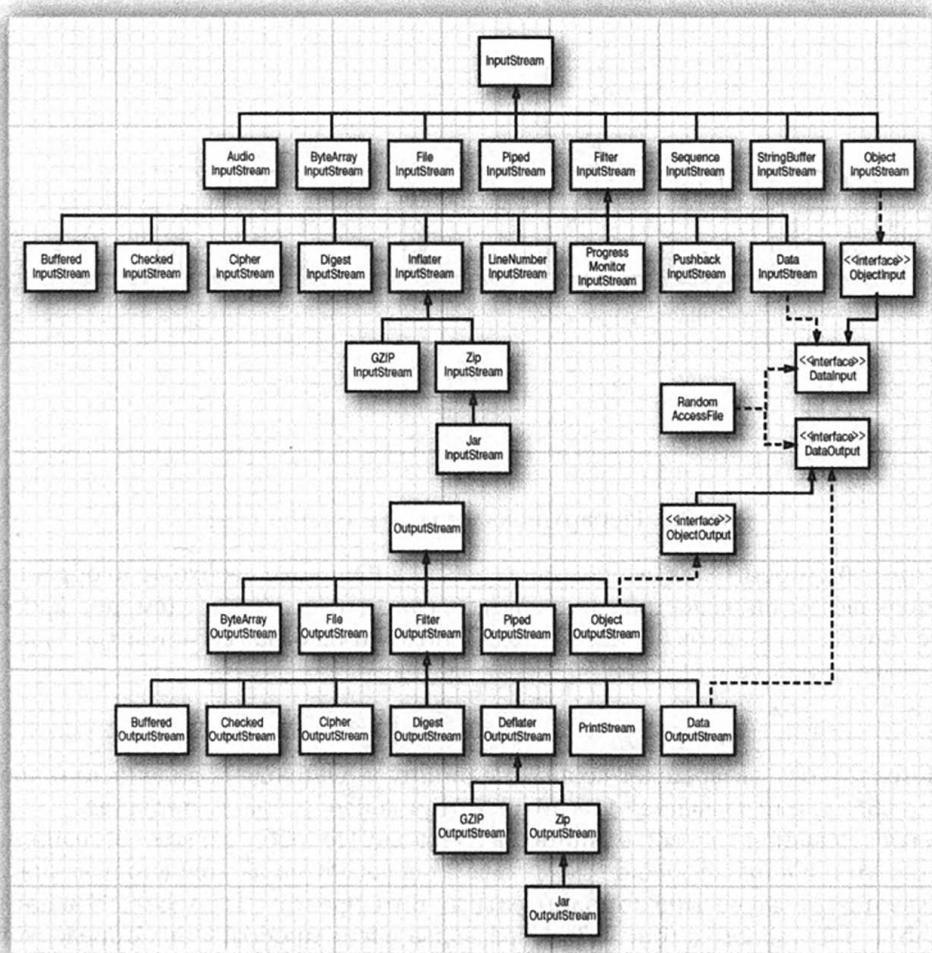


Рис. 2.1. Иерархия классов для потоков ввода-вывода

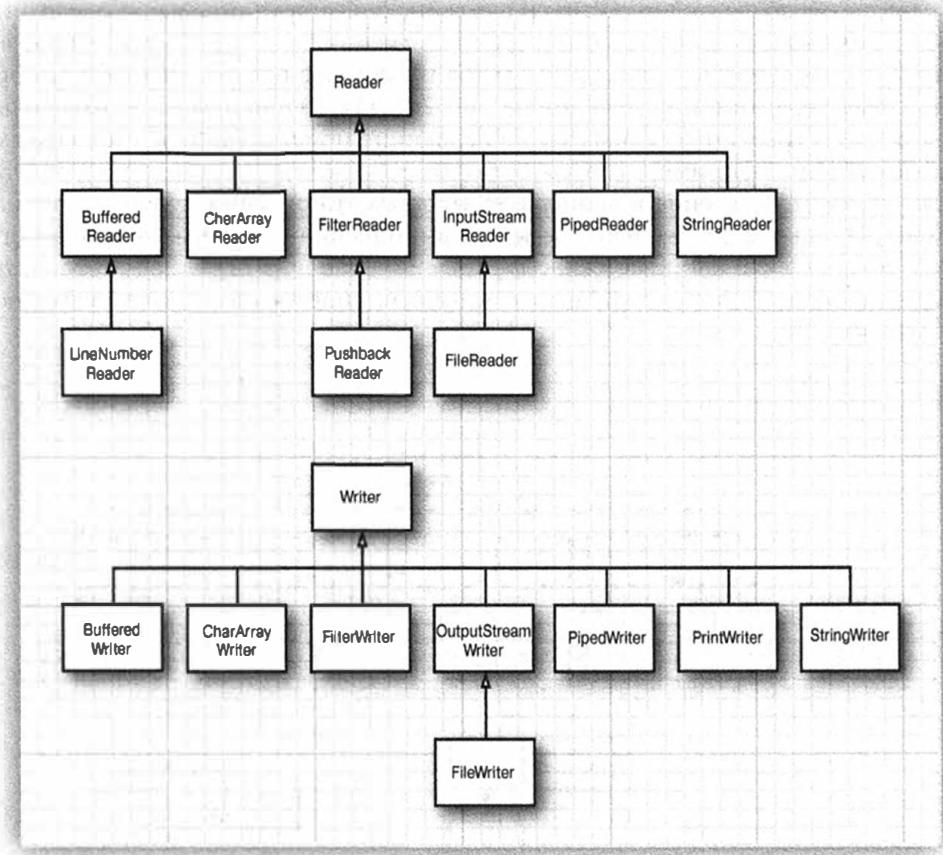


Рис. 2.2. Иерархия классов Reader и Writer

С другой стороны, для ввода-вывода текста в Юникоде необходимо обращаться к подклассам таких абстрактных классов, как Reader и Writer (см. рис. 2.2). Базовые методы из классов Reader и Writer похожи на базовые методы из классов InputStream и OutputStream:

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

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

Имеются также четыре дополнительных интерфейса: Closeable, Flushable, Readable и Appendable (рис. 2.3). Первые два очень просты: в них определяется единственный метод void close() throws IOException и void flush() соответственно. Классы InputStream, OutputStream, Reader и Writer реализуют интерфейс Closeable, а классы OutputStream и Writer — интерфейс Flushable.

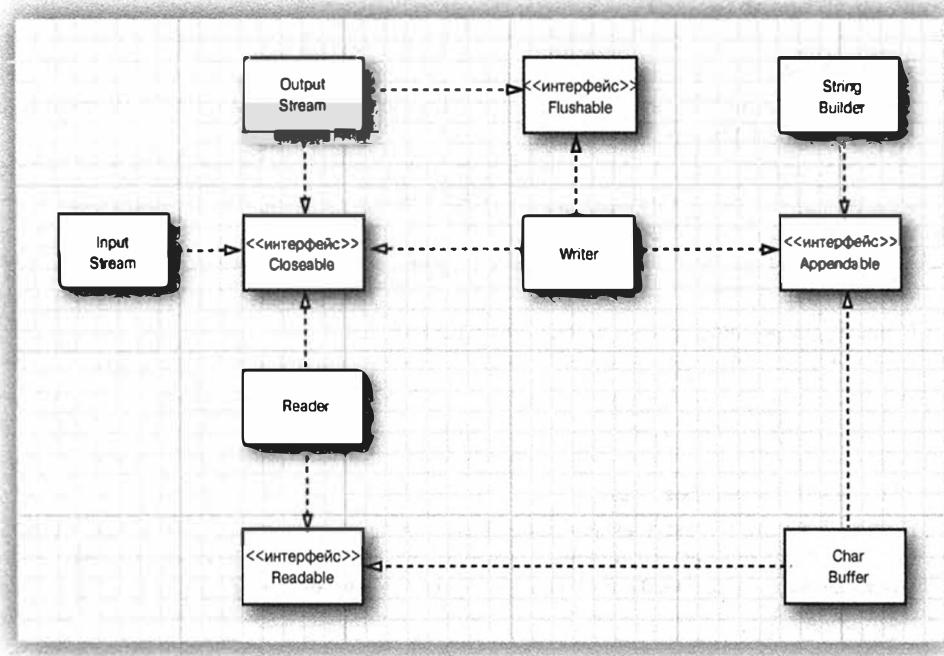


Рис. 2.3. Интерфейсы Closeable, Flushable, Readable и Appendable

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

В интерфейсе `Readable` имеется единственный метод `int read(CharBuffer cb)`, а в классе `CharBuffer` — методы для чтения и записи с последовательным и произвольным доступом. Этот класс представляет буфер в оперативной памяти или файл, отображаемый в памяти. (Подробнее об этом речь пойдет в разделе 2.6.2.)

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

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

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

java.io.Closeable 5.0

- **void close()**

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

java.io.Flushable 5.0

- **void flush()**

Очищает данный поток ввода-вывода типа **Flushable**.

java.lang.Readable 5.0

- **int read(CharBuffer cb)**

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

java.lang.Appendable 5.0

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

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

java.lang.CharSequence 1.4

- **char charAt(int index)**

Возвращает кодовую единицу по заданному индексу.

- **int length()**

Возвращает сведения об общем количестве кодовых единиц в данной последовательности.

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

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

- **String toString()**

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

2.1.3. Сочетание фильтров потоков ввода-вывода

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

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



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



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

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

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

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

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

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

В Java применяется искусственный механизм для разделения двух видов обвязностей. Одни потоки ввода-вывода (типа `FileInputStream` и поток ввода, возвращаемый методом `openStream()` из класса `URL`) могут извлекать байты из файлов и других экзотических мест, а другие потоки ввода-вывода (типа `DataInputStream`) — составлять эти байты в более полезные типы данных. Программирующему на Java остается только использовать их в нужном сочетании. Например, чтобы получить возможность читать числа из файла, достаточно

создать объект потока ввода типа `FileInputStream`, а затем передать его конструктору класса `DataInputStream`:

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

Если снова обратиться к рис. 2.1, то в иерархии классов для потоков ввода-вывода можно обнаружить классы `FilterInputStream` и `FilterOutputStream`. Подклассы этих классов служат для расширения функциональных возможностей потоков ввода-вывода, обрабатывающих байты.

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

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

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

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

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

Теперь можно прочитать сначала следующий байт с упреждением:

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

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

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

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

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

Безусловно, в библиотеках потоков ввода-вывода на других языках программирования такие полезные функции, как буферизация и чтение с упреждением, обеспечиваются автоматически, и поэтому в Java необходимость сочетать для их реализации потоковые фильтры доставляет лишние хлопоты. Но в то же время возможность сочетать и подбирать классы фильтров для создания действительно полезных последовательностей потоков ввода-вывода дает немалую свободу действий. Например, используя приведенную ниже последовательность потоков ввода, можно организовать чтение чисел из архивного файла, уплотненного в формате ZIP (рис. 2.4). (Более подробно о том, как манипулировать в Java файлами, уплотненными в формате ZIP, речь пойдет в разделе 2.3.3.)

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

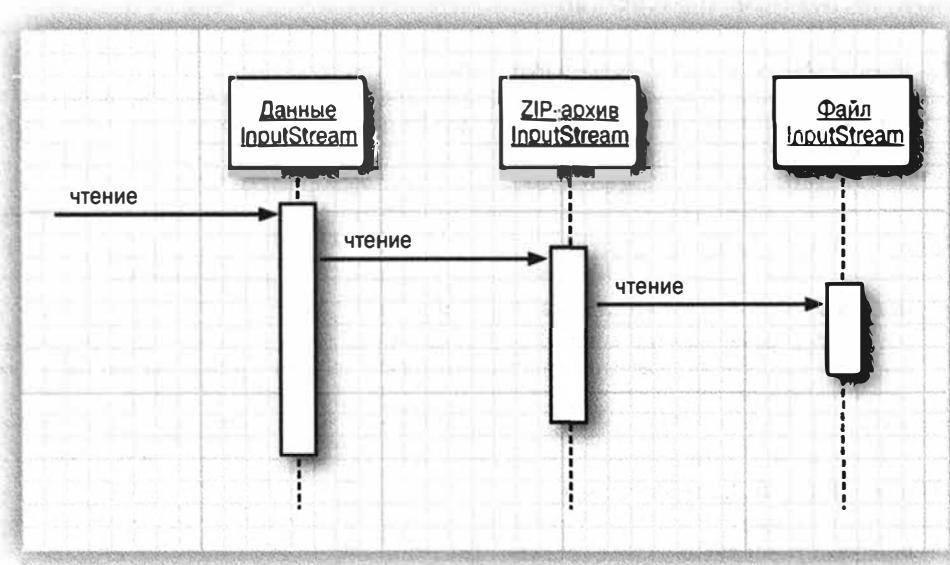


Рис. 2.4. Последовательность фильтруемых потоков

java.io.FileInputStream 1.0

- **FileInputStream(String name)**
- **FileInputStream(File file)**
- Создают новый поток ввода из файла, путь к которому указывается в символьной строке **name** или в объекте **file**. (Более подробно класс **File** описывается в конце этой главы.) Если указываемый путь не является абсолютным, он определяется относительно рабочего каталога, который был установлен при запуске виртуальной машины.

java.io.FileOutputStream 1.0

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

Создают новый поток вывода в файл, который указывается в символьной строке **name** или в объекте **file**. (Более подробно класс **File** описывается в конце этой главы.) Если параметр **append** принимает логическое значение **true**, существующий файл с таким же именем не удаляется, а данные добавляются в конце файла. В противном случае удаляется любой уже существующий файл с таким же именем.

java.io.BufferedInputStream 1.0

- **BufferedInputStream(InputStream in)**

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

java.io.BufferedOutputStream 1.0

- **BufferedOutputStream(OutputStream out)**

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

java.io.PushbackInputStream 1.0

- **PushbackInputStream(InputStream in)**
 - **PushbackInputStream(InputStream in, int size)**
- Создают поток ввода или вывода с однобайтовым буфером для чтения с упреждением или буфером указанного размера для возврата данных обратно в поток.

- **void unread(int b)**
Возвращает байт обратно в поток, чтобы он мог быть извлечен снова при последующем вызове для чтения.

Параметры:

b

Повторно читаемый байт

2.2. Ввод-вывод текста

При сохранении данных приходится выбирать между двоичным и текстовым форматом. Так, если целое число **1234** сохраняется в двоичном формате, оно записывается в виде следующей последовательности байтов: **00 00 04 D2** (в шестнадцатеричном представлении), а в текстовом формате — в виде символьной

строки "1234". Хотя ввод-вывод данных в двоичном формате осуществляется быстрее и эффективнее, тем не менее, они неудобочитаемы. Поэтому мы обсудим сначала ввод-вывод текстовых, а затем двоичных данных (в разделе 2.3).

При сохранении текстовых строк приходится учитывать конкретную кодировку символов. Так, если используется кодировка UTF-16, символьная строка "José" кодируется последовательностью байтов 00 4A 00 6F 00 73 00 E9 (в шестнадцатеричном представлении). Но во многих прикладных программах предполагается другая кодировка содержимого текстовых файлов. Так, в кодировке UTF-8, чаще всего употребляемой в Интернете, упомянутая выше символьная строка будет записана в виде последовательности байтов 4A 6F 73 C3 A9, где первые три буквы представлены без нулевых байтов, а последняя буква é — двумя байтами.

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

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

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

В этом потоке чтения вводимых данных предполагается, что в системе используется стандартная кодировка символов. В настольных операционных системах может применяться устаревшая кодировка вроде Windows 1252 или MacRoman. Но ничто не мешает выбрать любую другую кодировку, указав ее в конструкторе класса `InputStreamReader`, например, так, как показано ниже. Подробнее кодировки символов будут обсуждаться в разделе 2.2.4.

```
Reader in = new InputStreamReader(new FileInputStream("data.txt"),  
                                StandardCharsets.UTF_8);
```

2.2.1. Вывод текста

Для вывода текста лучше всего подходит класс `PrintWriter`, в котором имеются методы для вывода символьных строк и чисел в текстовом формате. В нем имеется даже удобный конструктор для вывода в файл. Так, операторы

```
PrintWriter out = new PrintWriter("employee.txt", "UTF-8");
```

и

```
PrintWriter out = new PrintWriter(  
    new FileOutputStream("employee.txt"), "UTF-8");
```

совершенно равнозначны.

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

Рассмотрим в качестве примера следующий фрагмент кода:

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

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

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

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

```
PrintWriter out = new PrintWriter(
    new OutputStreamWriter(
        new FileOutputStream("employee.txt"), "UTF-8"),
    true); // автоматическая очистка
```

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



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

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

java.io.PrintWriter 1.1

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

Создают новый экземпляр класса `PrintWriter`, выводящий данные в указанный поток записи.

java.io.PrintWriter 1.1

- **PrintWriter(OutputStream out)**
Создают новый экземпляр класса `PrintWriter` из существующего экземпляра класса `OutputStream`, получая необходимый промежуточный экземпляр класса `OutputStreamWriter`.
PrintWriter(String filename, String encoding)
- **PrintWriter(File file, String encoding)**
Создают новый экземпляр класса `PrintWriter` для записи данных в указанный файл, используя заданную кодировку.
- **void print(Object obj)**
Выводит объект в виде символьной строки, получаемой из метода `toString()`.
- **void print(String s)**
Выводит символьную строку в виде кодовых единиц Юникода.
- **void println(String s)**
Выводит символьную строку вместе с символом окончания строки. Очищает поток вывода, если он действует в режиме автоматической очистки.
- **void print(char[] s)**
Выводит все символы из указанного массива в виде кодовых единиц Юникода.
- **void print(char c)**
Выводит символ в виде кодовой единицы Юникода.
- **void print(int i)**
- **void print(long l)**
- **void print(float f)**
- **void print(double d)**
- **void print(boolean b)**
Выводят заданное значение в текстовом формате.
- **void printf(String format, Object... args)**
Выводит заданные значения так, как указано в форматирующей строке. О том, как задается форматирующая строка, см. в главе 3 первого тома настоящего издания.
- **boolean checkError()**
Возвращает логическое значение `true`, если возникла ошибка при форматировании или выводе данных. При возникновении ошибки поток вывода считается испорченным, а в результате всех вызовов метода `checkError()` возвращается логическое значение `true`.

2.2.2. Ввод текста

Для обработки произвольно вводимого текста проще всего воспользоваться классом `Scanner`, как неоднократно демонстрировалось в примерах кода из первого тома настоящего издания. Объект типа `Scanner` можно создать из любого потока ввода.

С другой стороны, прочитать короткий текст из файла в символьную строку можно следующим образом:

```
String content = new String(Files.readAllBytes(path), charset);
```

Но если требуется прочитать содержимое файла в виде последовательности строк, то необходимо сделать следующий вызов:

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

Если же файл крупный, строки можно обрабатывать по требованию в виде потока типа Stream<String>, как показано ниже.

```
try (Stream<String> lines = Files.lines(path, charset))
{
    . . .
}
```

В ранних версиях Java единственным возможным вариантом для обработки вводимых текстовых данных был класс BufferedReader. В этом классе имеется метод readLine(), возвращающий текстовую строку или пустое значение null, если больше нечего вводить. Следовательно, типичный цикл ввода текстовых данных выглядит следующим образом:

```
InputStream inputStream = . . .;
try (BufferedReader in = new BufferedReader(new InputStreamReader(
                    inputStream, StandardCharsets.UTF_8)))
{
    String line;
    while ((line = in.readLine()) != null)
    {
        // сделать что-нибудь со строкой
    }
}
```

Теперь в классе BufferedReader появился также метод lines(), возвращающий поток типа Stream<String>. Но, в отличие от класса Scanner, в классе BufferedReader отсутствуют методы для чтения числовых данных.

2.2.3. Сохранение объектов в текстовом формате

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

```
Harry Hacker|35500|1989-10-1
Carl Cracker|75000|1987-12-15
Tony Tester|38000|1990-03-15
```

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

```
public static void writeEmployee(PrintWriter out, Employee e)
{
    out.println(e.getName() + "!" + e.getSalary()
                + "!" + e.getHireDay());
}
```

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

```
public static Employee readEmployee(Scanner in)
{
    String line = in.nextLine();
    String[] tokens = line.split("\\|");
    String name = tokens[0];
    double salary = Double.parseDouble(tokens[1]);
    LocalDate hireDate = LocalDate.parse(tokens[2]);
    int year = hireDate.getYear();
    int month = hireDate.getMonthValue();
    int day = hireDate.getDayOfMonth();
    return new Employee(name, salary, year, month, day);
}
```

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

Весь исходный код данного примера программы представлен в листинге 2.1. А в приведенном ниже статическом методе сначала записывается длина массива, а затем каждая запись.

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

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

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

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

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

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

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

```
1 package textFile;
2
3 import java.io.*;
4 import java.time.*;
5 import java.util.*;
6
7 /**
8 * @version 1.14 2016-07-11
9 * @author Cay Horstmann
10 */
11 public class TextFileTest
12 {
13     public static void main(String[] args) throws IOException
14     {
15         Employee[] staff = new Employee[3];
16
17         staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
18         staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
19         staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
20
21         // сохранить записи обо всех сотрудниках в файле employee.dat
22         try (PrintWriter out =
23             new PrintWriter("employee.dat", "UTF-8"))
24         {
25             writeData(staff, out);
26         }
27
28         // извлечь все записи в новый массив
29         try (Scanner in = new Scanner(
30             new FileInputStream("employee.dat"), "UTF-8"))
31         {
32             Employee[] newStaff = readData(in);
33
34             // вывести вновь прочитанные записи о сотрудниках
35             for (Employee e : newStaff)
36                 System.out.println(e);
37         }
38     }
39
40 /**
41 * Записывает данные обо всех сотрудниках из
42 * массива в поток записи выводимых данных
43 * @param employees Массив записей о сотрудниках
44 * @param out Поток записи выводимых данных
45 */
46 private static void writeData(
47     Employee[] employees, PrintWriter out) throws IOException
48 {
49     // записать количество сотрудников
50     out.println(employees.length);
51
52     for (Employee e : employees)
53         writeEmployee(out, e);
54 }
55
56 /**
```

```
57 * Читает записи о сотрудниках из потока сканирования в массив
58 * @param in Поток сканирования вводимых данных
59 * @return Массив записей о сотрудниках
60 */
61 private static Employee[] readData(Scanner in)
62 {
63     // извлечь размер массива
64     int n = in.nextInt();
65     in.nextLine(); // consume newline
66
67     Employee[] employees = new Employee[n];
68     for (int i = 0; i < n; i++)
69     {
70         employees[i] = readEmployee(in);
71     }
72     return employees;
73 }
74
75 /**
76 * Направляет данные о сотрудниках в поток
77 * записи выводимых данных
78 * @param out Поток записи выводимых данных
79 */
80 public static void writeEmployee(PrintWriter out, Employee e)
81 {
82     out.println(e.getName() + "|" + e.getSalary()
83                 + "|" + e.getHireDay());
84 }
85
86 /**
87 * Считывает данные о сотрудниках из буферизованного
88 * потока чтения вводимых данных
89 * @param in Поток чтения/сканирования вводимых данных
90 */
91 public static Employee readEmployee(Scanner in)
92 {
93     String line = in.nextLine();
94     String[] tokens = line.split("\\|");
95     String name = tokens[0];
96     double salary = Double.parseDouble(tokens[1]);
97     LocalDate hireDate = LocalDate.parse(tokens[2]);
98     int year = hireDate.getYear();
99     int month = hireDate.getMonthValue();
100    int day = hireDate.getDayOfMonth();
101    return new Employee(name, salary, year, month, day);
102 }
103 }
```

2.2.4. Кодировки символов

Потоки ввода-вывода представляют собой последовательности байтов, но зачастую приходится обрабатывать текст, т.е. последовательности символов. В таком случае имеет значение, каким образом символы кодируются в байты.

Для кодирования символов в Java применяется стандарт, называемый **Юникодом** (Unicode). Каждый символ, или так называемая *кодовая точка*, представлен

в Юникоде 21-разрядным целым числом. Имеются разные кодировки символов — способы упаковки 21-разрядных целых чисел в байты.

Чаще всего применяется кодировка UTF-8, в которой каждая кодовая точка в Юникоде кодируется последовательностью от одного до четырех байтов (табл. 2.1). Преимущество кодировки UTF-8 состоит в том, что символы из традиционного набора в коде ASCII, куда входят все буквы латинского и английского алфавитов, занимают только один байт.

Таблица 2.1. Кодировка UTF-8

Диапазон символов	Кодировка
0...7F	0a ₆ a ₅ a ₄ a ₃ a ₂ a ₁ a ₀
80...7FF	110a ₁₀ a ₉ a ₈ a ₇ a ₆ 10a ₅ a ₄ a ₃ a ₂ a ₁ a ₀
800...FFFF	1110a ₁₅ a ₁₄ a ₁₃ a ₁₂ 10a ₁₁ a ₁₀ a ₉ a ₈ a ₇ a ₆ 10a ₅ a ₄ a ₃ a ₂ a ₁ a ₀
10000...10FFFF	11110a ₂₀ a ₁₉ a ₁₈ 10a ₁₇ a ₁₆ a ₁₅ a ₁₄ a ₁₃ a ₁₂ 10a ₁₁ a ₁₀ a ₉ a ₈ a ₇ a ₆ 10a ₅ a ₄ a ₃ a ₂ a ₁ a ₀

Еще одной распространенной является кодировка UTF-16, в которой каждая кодовая точка в Юникоде кодируется одним или двумя 16-разрядными значениями (табл. 2.2). Такая кодировка применяется в символьных строках Java. На самом деле имеются две формы кодировки UTF-16: с обратным и прямым порядком следования байтов. Рассмотрим в качестве примера 16-разрядное значение **0x2122**. В формате с обратным порядком байтов первым следует старший байт **0x21**, а за ним — младший байт **0x22**. А в формате с прямым порядком байтов все происходит наоборот: сначала следует младший байт **0x22**, а затем старший байт **0x21**. Для обозначения используемого порядка следования байтов файл может начинаться с соответствующей метки в виде 16-разрядного значения **0xFEFF**. С помощью этой метки пользователь может определить порядок следования байтов и отбросить ее.

Таблица 2.2. Кодировка UTF-16

Диапазон символов	Кодировка
0...FFFF	a ₁₅ a ₁₄ a ₁₃ a ₁₂ a ₁₁ a ₁₀ a ₉ a ₈ a ₇ a ₆ a ₅ a ₄ a ₃ a ₂ a ₁ a ₀
10000...10FFFF	110110b ₁₉ b ₁₈ b ₁₇ b ₁₆ a ₁₅ a ₁₄ a ₁₃ a ₁₂ a ₁₁ a ₁₀ 110111a ₉ a ₈ a ₇ a ₆ a ₅ a ₄ a ₃ a ₂ a ₁ a ₀ , где b ₁₉ b ₁₈ b ₁₇ b ₁₆ = a ₂₀ a ₁₉ a ₁₈ a ₁₇ a ₁₆ - 1



Внимание! В некоторых прикладных программах, в том числе в текстовом редакторе Microsoft Notepad, метка порядка следования байтов вводится в начале файлов, содержимое которых представлено в кодировке UTF-8. Очевидно, что это излишне, поскольку в кодировке UTF-8 вопросы, связанные с порядком следования байтов, не возникают. Тем не менее стандарт на Юникод допускает наличие такой метки и даже считает это целесообразным, поскольку она разрешает всякие сомнения по поводу кодировки. При чтении содержимого файла в кодировке UTF-8 такая метка должна удаляться. К сожалению, в Java этого не делается, а в отчетах об устранимых программных ошибках напротив данной ошибки стоит метка “не устранится”. Поэтому любой начальный код \uFEFF, обнаруживаемый при вводе данных, придется удалять вручную.

Помимо упомянутых выше кодировок UTF, имеются частичные кодировки, охватывающие диапазон символов, пригодных для конкретного круга

пользователей. Например, кодировка по стандарту ISO 8859-1 определяет однобайтовый код, включающий в себя символы с ударениями, применяемые в западноевропейских языках, а кодировка Shift-JIS — код переменной длины для японских символов. Немалое число подобных кодировок по-прежнему широко распространено.

Надежного способа автоматически выявить кодировку символов в потоке ввода байтов не существует. В некоторых методах из прикладного программного интерфейса API допускается применение “набора символов по умолчанию” — кодировки символов, которая считается наиболее предпочтительной в операционной системе компьютера. Но применяется ли та же самая кодировка и в источнике байтов? Ведь эти байты вполне могут поступать из разных частей света. Следовательно, кодировка символов должна всегда указываться явно. Так, при чтении веб-страницы следует проверять заголовок Content-Type.



На заметку! Кодировка, принятая на конкретной платформе, возвращается статическим методом `Charset.defaultCharset()`. А статический метод `Charset.availableCharsets()` возвращает все имеющиеся экземпляры класса `Charset`, преобразованные из канонических имен в объекты типа `Charset`.



Внимание! В реализации библиотеки Java от компании Oracle имеется системное свойство `file.encoding` для переопределения кодировки символов, принятой на конкретной платформе по умолчанию. Но это свойство не поддерживается официально и не последовательно согласуется со всеми частями реализации библиотеки Java от компании Oracle. Поэтому устанавливать его не следует.

В классе `StandardCharsets` имеются следующие статические переменные типа `Charset` для тех кодировок, которые должны поддерживаться на каждой виртуальной машине Java:

```
StandardCharsets.UTF_8  
StandardCharsets.UTF_16  
StandardCharsets.UTF_16BE  
StandardCharsets.UTF_16LE  
StandardCharsets.ISO_8859_1  
StandardCharsets.US_ASCII
```

Чтобы получить объект типа `Charset` для другой кодировки, достаточно вызвать статический метод `forName()` следующим образом:

```
Charset shiftJIS = Charset.forName("Shift-JIS");
```

Для ввода-вывода текста следует пользоваться объектом типа `Charset`. В следующем примере кода показано, как превратить массив байтов в символьную строку:

```
String str = new String(bytes, StandardCharsets.UTF_8);
```



Совет! При вызове некоторых методов допускается указывать кодировку символов с помощью объекта типа `Charset` или символьной строки. Чтобы не утруждать себя указанием правильного написания кодировки, достаточно выбрать подходящую константу из класса `StandardCharsets`. Так, выражение `new String(bytes, "UTF_8")` неприемлемо и вызовет ошибку.



Внимание! В одних методах и конструкторах, например, в конструкторе `String(byte[])`, используется кодировка, принятая на платформе по умолчанию, если не указано иное. А в других методах и конструкторах, например, в конструкторе `Files.readAllLines()`, применяется кодировка UTF-8.

2.3. Чтение и запись двоичных данных

Текстовый формат удобен для тестирования и отладки прикладного кода, поскольку он удобочитаем. Но он не столь эффективен для передачи данных, как двоичный формат. Поэтому в последующих разделах поясняется, каким образом организует ввод и вывод двоичных данных.

2.3.1. Интерфейсы `DataInput` и `DataOutput`

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

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

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



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

Это может стать причиной серьезных осложнений. Например, данные сохраняются в исходном файле программы на С или C++ именно так, как это делает процессор. Вследствие этого

перемещение даже простейших файлов данных с одной платформы на другую превращается в совсем не простую задачу. А в Java все значения записываются в формате с обратным порядком следования байтов от старшего к младшему независимо от типа процессора, что, соответственно, делает файлы данных в Java независящими от используемой платформы.

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

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

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

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

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

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

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

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

java.io.DataInput 1.0

- `boolean readBoolean()`
- `byte readByte()`
- `char readChar()`
- `double readDouble()`

java.io.DataInput 1.0 (окончание)

- **float readFloat()**

- **int readInt()**

- **long readLong()**

- **short readShort()**

Считывают значение заданного типа.

- **void readFully(byte[] b)**

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

Параметры:

b

Массив, в который должны быть
считаны байты

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

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

Параметры:

b

Массив, в который должны быть
считаны байты

off

Начальное смещение данных

len

Максимальное количество
считываемых байтов

- **String readUTF()**

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

- **int skipBytes(int n)**

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

Параметры:

n

Количество пропускаемых байтов

java.io.DataOutput 1.0

- **void writeBoolean(boolean b)**

- **void writeByte(int b)**

- **void writeChar(int c)**

- **void writeDouble(double d)**

- **void writeFloat(float f)**

- **void writeInt(int i)**

- **void writeLong(long l)**

- **void writeShort(int s)**

Записывают значение заданного типа.

- **void writeChars(String s)**

Записывает все символы из строки.

- **void writeUTF(String s)**

Записывает символьную строку в "модифицированном" формате UTF-8

2.3.2. Файлы с произвольным доступом

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

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

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

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

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

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

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

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

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

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

Целочисленные значения и числовые значения с плавающей точкой имеют фиксированный размер в двоичном формате, тогда как с символьными строками

дело обстоит немного сложнее. Для записи и чтения символьных строк фиксированного размера придется ввести два вспомогательных метода. В частности, приведенный ниже метод `writeFixedString()` записывает указанное количество кодовых единиц, отсчитывая от начала символьной строки. (Если кодовых единиц слишком мало, символьная строка дополняется нулевыми значениями.)

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

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

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

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

```
DataIO.writeFixedString(e.getName(), Employee.NAME_SIZE, out);
out.writeDouble(e.getSalary());
LocalDate hireDay = e.getHireDay();
out.writeInt(hireDay.getYear());
out.writeInt(hireDay.getMonthValue());
out.writeInt(hireDay.getDayOfMonth());
```

Обратное чтение данных выполняется так же просто:

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

```
int m = in.readInt();
int d = in.readInt();
```

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

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

Программа из листинга 2.2 делает три записи в файле данных, а затем читает их оттуда в обратном порядке. Для эффективного выполнения данного процесса требуется произвольный доступ к файлу, а в данном случае — доступ сначала к третьей записи.

Листинг 2.2. Исходный код из файла randomAccess/RandomAccessTest.java

```
1 package randomAccess;
2
3 import java.io.*;
4 import java.util.*;
5 import java.time.*;
6
7 /**
8  * @version 1.13 2016-07-11
9  * @author Cay Horstmann
10 */
11 public class RandomAccessTest
12 {
13     public static void main(String[] args) throws IOException
14     {
15         Employee[] staff = new Employee[3];
16
17         staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
18         staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
19         staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
20
21         try (DataOutputStream out = new DataOutputStream(
22                 new FileOutputStream("employee.dat")))
23         {
24             // сохранить все записи о сотрудниках в файле employee.dat
25             for (Employee e : staff)
26                 writeData(out, e);
27         }
28
29         try (RandomAccessFile in =
30                 new RandomAccessFile("employee.dat", "r"))
31         {
32             // извлечь все записи в новый массив
33
34             // определить размер массива
35             int n = (int)(in.length() / Employee.RECORD_SIZE);
36             Employee[] newStaff = new Employee[n];
37         }
```

```

38     // прочитать записи о сотрудниках в обратном порядке
39     for (int i = n - 1; i >= 0; i--)
40     {
41         newStaff[i] = new Employee();
42         in.seek(i * Employee.RECORD_SIZE);
43         newStaff[i] = readData(in);
44     }
45
46     // вывести вновь прочитанные записи о сотрудниках
47     for (Employee e : newStaff)
48         System.out.println(e);
49     }
50 }
51
52 /**
53 * Записывает сведения о сотрудниках в поток вывода данных
54 * @param out Поток вывода данных
55 * @param e Сотрудник
56 */
57 public static void writeData(DataOutput out, Employee e)
58     throws IOException
59 {
60     DataIO.writeFixedString(e.getName(), Employee.NAME_SIZE, out);
61     out.writeDouble(e.getSalary());
62
63     LocalDate hireDay = e.getHireDay();
64     out.writeInt(hireDay.getYear());
65     out.writeInt(hireDay.getMonthValue());
66     out.writeInt(hireDay.getDayOfMonth());
67 }
68
69 /**
70 * Читает сведения о сотрудниках из потока ввода данных
71 * @param in Поток ввода данных
72 * @return Возвращает сотрудника
73 */
74 public static Employee readData(DataInput in) throws IOException
75 {
76     String name = DataIO.readFixedString(Employee.NAME_SIZE, in);
77     double salary = in.readDouble();
78     int y = in.readInt();
79     int m = in.readInt();
80     int d = in.readInt();
81     return new Employee(name, salary, y, m - 1, d);
82 }
83 }

```

java.io.RandomAccessFile 1.0

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

Параметры: **file** Открываемый файл

mode **r** — режим только для чтения;
rw — режим чтения и записи;

java.io.RandomAccessFile 1.0 (окончание)

rws — режим чтения и записи с синхронным записыванием на диск данных и метаданных при каждом обновлении;

rwd — режим чтения и записи с синхронным записыванием на диск только данных

- **long getFilePointer()**
Возвращает сведения о текущем местоположении указателя файла.
- **void seek(long pos)**
Устанавливает указатель файла на позицию *pos* от начала файла.
- **long length()**
Возвращает длину файла в байтах.

2.3.3. ZIP-архивы

В ZIP-архивах можно хранить один файл или больше (как правило) в уплотненном формате. У каждого ZIP-архива имеется заголовок, содержащий сведения вроде имени файла или применявшегося для него алгоритма сжатия. В Java для чтения ZIP-архивов служит класс ZipInputStream. В каждом таком архиве всегда требуется просматривать отдельные записи. Метод getNextEntry() возвращает описывающий запись объект типа ZipEntry. Чтобы получить поток ввода для чтения записи из архива, эту запись следует передать методу getInputStream(). Далее вызывается метод closeEntry() для перехода к чтению следующей записи. Ниже приведен типичный фрагмент кода для чтения содержимого ZIP-файла.

```
ZipInputStream zin =
    new ZipInputStream(new FileInputStream(zipname));
ZipEntry entry;
while ((entry = zin.getNextEntry()) != null)
{
    InputStream in = zin.getInputStream(entry);
    прочитать содержимое потока ввода in
    zin.closeEntry();
}
zin.close();
```

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

```

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

```



На заметку! Архивные JAR-файлы [см. главу 13 первого тома настоящего издания] представляют собой те же ZIP-файлы, но только они содержат записи несколько иного вида, называемые манифестами. Для чтения и записи манифестов служат классы `JarInputStream` и `JarOutputStream`.

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



На заметку! Далее, в разделе 2.5.8, поясняется, как осуществить доступ к ZIP-архиву без специального прикладного программного интерфейса API, используя класс `FileSystem`, внедренный в версии Java SE 7.

java.util.zip.ZipInputStream 1.1

- **ZipInputStream(InputStream in)**

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

- **ZipEntry getNextEntry()**

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

- **void closeEntry()**

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

java.util.zip.ZipOutputStream 1.1

- **ZipOutputStream(OutputStream out)**

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

java.util.zip.ZipInputStream 1.1 (окончание)

- **ZipOutputStream(OutputStream out)**

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

- **void putNextEntry(ZipEntry ze)**

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

- **void closeEntry()**

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

- **void setLevel(int level)**

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

Параметры: **level** Степень сжатия от 0

[NO_COMPRESSION] до 9

(BEST_COMPRESSION)

- **void setMethod(int method)**

- Устанавливает алгоритм сжатия по умолчанию для любых записей, направляемых в поток вывода типа **ZipOutputStream**, без указания конкретного алгоритма сжатия данных.

Параметры: **method** Алгоритм сжатия данных:

DEFLATED или **STORED**

java.util.zip.ZipEntry 1.1

- **ZipEntry(String name)**

Создает запись в Zip-архиве с заданным именем.

Параметры: **name** Имя записи

- **long getCrc()**

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

- **String getName()**

Возвращает имя данной записи.

- **long getSize()**

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

- **boolean isDirectory()**

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

java.util.zip.ZipEntry 1.1 (окончание)

- **void setMethod(int method)**

Задает алгоритм сжатия записей.

Параметры: **method**

Алгоритм сжатия записей:

DEFLATED или **STORED**

- **void setSize(long size)**

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

Параметры: **size**

Размер данной записи без сжатия

- **void setCrc(long crc)**

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

Параметры: **crc**

Контрольная сумма данной записи.

java.util.zip.ZipFile 1.1

- **ZipFile(String name)**

- **ZipFile(File file)**

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

- **Enumeration entries()**

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

- **ZipEntry getEntry(String name)**

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

Параметры: **name** Имя записи

- **InputStream getInputStream(ZipEntry ze)**

Возвращает поток ввода типа **InputStream** для указанной записи.

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

- **String getName()**

Возвращает путь к данному ZIP-архиву.

2.4. Потоки ввода-вывода и сериализация объектов

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

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

2.4.1. Сохранение и загрузка сериализуемых объектов

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

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

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

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

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

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

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

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

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

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

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



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

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

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

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

Теперь каждый объект типа `Manager` будет содержать ссылку на объект типа `Employee`, описывающий секретаря. Безусловно, два руководителя вполне могут пользоваться услугами одного и того же секретаря, как показано на рис. 2.5 и в следующем фрагменте кода:

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

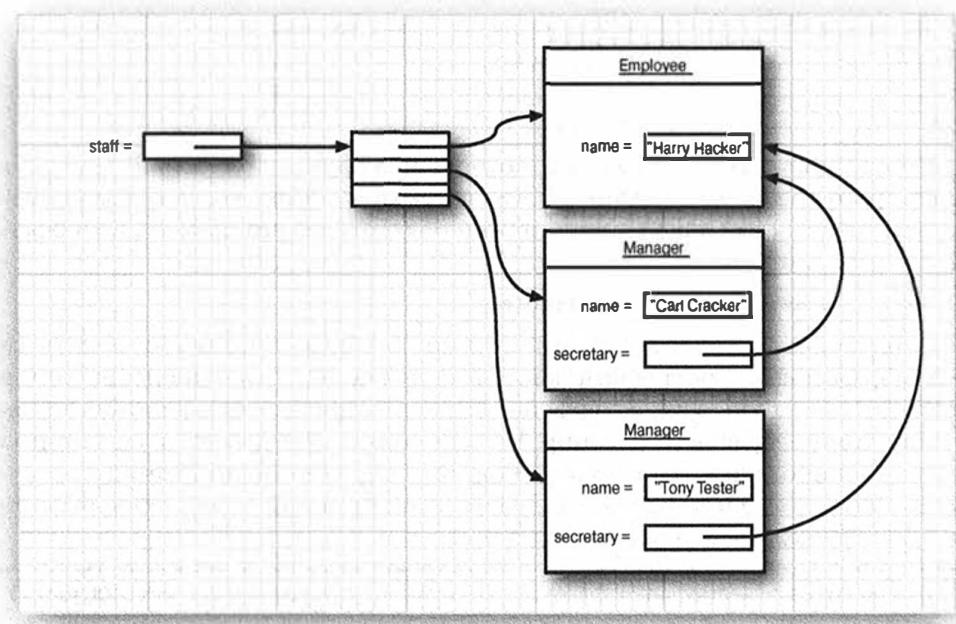


Рис. 2.5. Два руководителя могут совместно пользоваться услугами одного и того же сотрудника в качестве секретаря

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

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

1. Серийный (т.е. порядковый) номер связывается с каждой встречающейся ссылкой на объект, как показано на рис. 2.6.
2. Если ссылка на объект встречается впервые, данные из этого объекта сохраняются в потоке ввода-вывода объектов.
3. Если же данные были ранее сохранены, просто добавляется метка "same as previously saved object with serial number x" (совпадает с объектом, сохраненным ранее под серийным номером x).

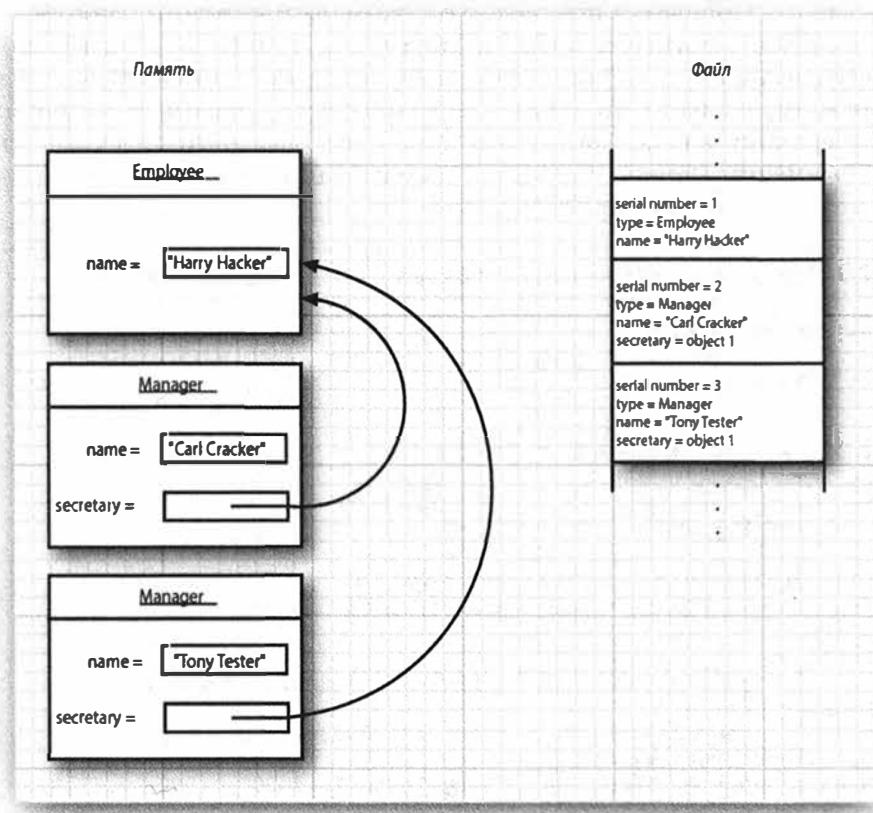


Рис. 2.6. Пример сериализации объектов

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

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



На заметку! В этой главе демонстрируется, каким образом можно применять механизм сериализации для сохранения объектов в файле на диске и извлечения их в точном соответствии с тем, как они сохранялись. Другой очень важной областью применения данного механизма является передача коллекции объектов по сетевому соединению на другой компьютер. Исходные адреса ячеек памяти не имеют никакого значения при взаимодействии с другим процессором, как и при обращении к файлу. Заменяя адреса ячеек памяти серийными номерами, механизм сериализации делает вполне возможным перенос коллекций объектов с одной машины на другую.

В листинге 2.3 приведен исходный код примера программы, способной сохранять и перезагружать сеть объектов типа Employee и Manager (некоторые из руководителей пользуются услугами общего сотрудника в качестве секретаря). Обратите внимание на то, что объект секретаря остается однозначным после повторной перезагрузки, т.е. когда сотрудник, представленный объектом, хранящимся в элементе массива newStaff[1], получает повышение, это отражается в полях secretary объектов типа Manager.

Листинг 2.3. Исходный код из файла objectStream/ObjectStreamTest.java

```
1 package objectStream;
2
3 import java.io.*;
4
5 /**
6  * @version 1.10 17 Aug 1998
7  * @author Cay Horstmann
8 */
9 class ObjectStreamTest
10 {
11     public static void main(String[] args)
12         throws IOException, ClassNotFoundException
13     {
14         Employee harry =
15             new Employee("Harry Hacker", 50000, 1989, 10, 1);
16         Manager carl =
17             new Manager("Carl Cracker", 80000, 1987, 12, 15);
18         carl.setSecretary(harry);
19         Manager tony = new Manager("Tony Tester", 40000, 1990, 3, 15);
20         tony.setSecretary(harry);
21
22         Employee[] staff = new Employee[3];
23
24         staff[0] = carl;
```

```

25     staff[1] = harry;
26     staff[2] = tony;
27
28     // сохранить записи обо всех сотрудниках в файле employee.dat
29     try (ObjectOutputStream out = new ObjectOutputStream(
30         new FileOutputStream("employee.dat")))
31     {
32         out.writeObject(staff);
33     }
34
35     try (ObjectInputStream in = new ObjectInputStream(
36         new FileInputStream("employee.dat")))
37     {
38         // извлечь все записи в новый массив
39
40         Employee[] newStaff = (Employee[]) in.readObject();
41
42         // поднять зарплату секретарю
43         newStaff[1].raiseSalary(10);
44
45         // вывести вновь прочитанные записи о сотрудниках
46         for (Employee e : newStaff)
47             System.out.println(e);
48     }
49 }
50 }
```

`java.io.ObjectOutputStream` 1.1

- **`ObjectOutputStream(OutputStream out)`**

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

- **`void writeObject(Object obj)`**

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

`java.io.ObjectInputStream` 1.1

- **`ObjectInputStream(InputStream in)`**

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

- **`Object readObject()`**

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

2.4.2. Представление о формате файлов для сериализации объектов

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

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

74 Двухбайтовое число, обозначающее длину в файле

Символы

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

74 00 05 Harry

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

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

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

Алгоритм SHA позволяет быстро получить "отпечаток" с большого блока данных. Этот "отпечаток" всегда представляет собой 20-байтовый пакет данных, каким бы ни был размер исходных данных. Он создается в результате выполнения над данными некоторой искусно составленной последовательности поразрядных операций, что дает практически полную уверенность, что при любом изменении данных непременно изменится и сам "отпечаток". (Подробнее с алгоритмом SHA можно ознакомиться в книге *Cryptography and Network Security: Principles and Practice, Seventh Edition* Уильяма Столингса (William Stallings; издательство Prentice Hall, 2016 г.) Но в механизме сериализации для "отпечатка" класса используются

только первые 8 байтов кода SHA. И тем не менее это гарантирует, что при изменении полей данных или методов изменится и “отпечаток” класса.

При чтении объекта его “отпечаток” сравнивается с текущим “отпечатком” класса. Если они не совпадают, это означает, что после записи объекта определение класса изменилось, а следовательно, генерируется исключение. На практике классы постепенно усовершенствуются, и поэтому от прикладной программы, возможно, потребуется способность читать прежние версии объектов. Более подробно данный вопрос обсуждается далее, в разделе 2.4.5.

Идентификатор класса сохраняется в следующей последовательности.

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

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

```
static final byte SC_WRITE_METHOD = 1;
// в данном классе имеется метод writeObject(),
// записывающий дополнительные данные
static final byte SC_SERIALIZABLE = 2;
// в данном классе реализуется интерфейс Serializable
static final byte SC_EXTERNALIZABLE = 4;
// в данном классе реализуется интерфейс Externalizable
```

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

Каждый дескриптор поля данных имеет следующий формат.

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

Код типа может принимать одно из следующих значений.

- | | |
|---|---------|
| B | byte |
| C | char |
| D | double |
| F | float |
| I | int |
| J | long |
| L | объект |
| S | short |
| Z | boolean |
| [| МАССИВ |

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

D 00 06 salary

В качестве примера ниже полностью показан дескриптор класса Employee.

72 00 08 Employee

E6 D2 86 7D AE AC 18 1B 02	Отпечаток и флаги
00 03	Количество полей экземпляра
D 00 06 salary	Тип и имя поля экземпляра
L 00 07 hireDay	Тип и имя поля экземпляра
74 00 10 Ljava/util/Date;	Имя класса для поля экземпляра: Date
L 00 04 name	Тип и имя поля экземпляра
74 00 12 Ljava/lang/String;	Имя класса для поля экземпляра: String
78	Конечный маркер
70	Суперкласс отсутствует

Эти дескрипторы получаются довольно длинными. Поэтому если дескриптор одного и того же класса снова потребуется в файле, то для этой цели используется следующая сокращенная форма:

71 4-байтовый порядковый номер

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

73 Дескриптор класса

Данные объекта

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

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

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

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

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

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

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

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

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

Ниже приведен снабженный комментариями результат вывода из программы `ObjectStreamTest`, представленной в листинге 2.3. По желанию вы можете запустить эту программу на выполнение, посмотреть результат вывода данных в шестнадцатеричном виде из памяти в файл `employee.dat` и сравнить его с приведенным ниже результатом. Наиболее важные строки находятся ближе к концу и демонстрируют ссылку на сохраненный ранее объект.

AC ED 00 05	Заголовок файла
75	Массив staff (порядковый номер #1)
72 00 0B [LEmployee ;	Новый класс, длина строки, имя класса
FC BF 36 11 C5 91 11 C7 02	Employee [] (порядковый номер #0)
00 00	Отпечаток и флаги
78	Количество полей экземпляра
70	Конечный маркер
00 00 00 03	Суперкласс отсутствует
73	Количество записей в массиве
72 00 07 Manager	staff [0] — новый объект (порядковый номер #7)
36 06 AE 13 63 8F 59 B7 02	Новый класс, длина строки, имя класса (порядковый номер #2)
00 01	Отпечаток и флаги
L 00 09 secretary	Количество полей данных
74 00 0A [LEmployee ;	Тип и имя поля экземпляра
78	Имя класса для поля экземпляра — String (порядковый номер #3)
72 00 08 Employee	Конечный маркер
E6 D2 86 7D AE AC 18 1B 02	Суперкласс — новый класс, длина строки, имя класса (порядковый номер #4)
00 03	Отпечаток и флаги
D 00 06 salary	Количество полей экземпляра
L 00 07 hireDay	Тип и имя поля экземпляра
74 00 10 [Ljava/util/Date ;	Тип и имя поля экземпляра
L 00 04 name	Имя класса для поля экземпляра — String (порядковый номер #5)
74 00 12 [Ljava/lang/String ;	Тип и имя поля экземпляра
78	Имя класса для поля экземпляра — String (порядковый номер #6)
70	Конечный маркер
40 F3 88 00 00 00 00 00	Суперкласс отсутствует
73	Значение поля salary — double
72 00 0E java.util.Date	Значение поля secretary — новый объект (порядковый номер #9)
68 6A 81 01 4B 59 74 19 03	Новый класс, длина строки, имя класса (порядковый номер #8)
00 00	Отпечаток и флаги
78	Переменные экземпляра отсутствуют
77 08	Конечный маркер
00 00 00 83 E9 39 E0 00	Внешнее хранилище, количество байтов
78	Дата
74 00 0C Carl Cracker	Конечный маркер
	Значение поля name — String (порядковый номер #10)

73		Значение поля secretary — новый объект (порядковый номер #11)
71 00 7E 00 04		Существующий класс (использовать порядковый номер #4)
40 E8 6A 00 00 00 00 00		Значение поля salary — double
73		Значение поля hireDay — новый объект (порядковый номер #15)
71 00 7E 00 08		Существующий класс (использовать порядковый номер #8)
77 08		Внешнее хранилище, количество байтов
00 00 00 91 1B 4E B1 80		Дата
78		Конечный маркер
74 00 0C Harry Hacker		Значение поля name — String (порядковый номер #13)
71 00 7E 00 0B		staff[1] — существующий объект (использовать порядковый номер #11)
73		staff[2] — новый объект (порядковый номер #14)
71 00 7E 00 02		Существующий класс (использовать порядковый номер #2)
40 E3 88 00 00 00 00 00		Значение поля salary — double
73		Значение поля hireDay — новый объект (порядковый номер #15)
71 00 7E 00 08		Существующий класс (использовать порядковый номер #8)
77 08		Внешнее хранилище, количество байтов
00 00 00 94 6D 3E EC 00 00		Дата
78		Конечный маркер
74 00 0B Tony Tester		Значение поля name — String (порядковый номер #16)
71 00 7E 00 0B		Значение поля secretary — существующий объект (использовать порядковый номер #11)

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

Самое главное — запомнить следующее.

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

2.4.3. Видоизменение исходного механизма сериализации

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

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

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

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

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

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

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

В методе `readObject()` выполняется обратный процесс:

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

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

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

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

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

```
public void readExternal(ObjectInput s)
    throws IOException
{
    name = s.readUTF();
    salary = s.readDouble();
    hireDay = new Date(s.readLong());
}

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



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

2.4.4. Сериализация однозлементных множеств и типизированных перечислений

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

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

```
public class Orientation
{
    public static final Orientation HORIZONTAL = new Orientation(1);
    public static final Orientation VERTICAL = new Orientation(2);

    private int value;

    private Orientation(int v) { value = v; }
}
```

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

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

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

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

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

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

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

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

2.4.5. Контроль версий

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

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

`serialver Employee`

даст такой результат:

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

Если же запустить утилиту `serialver` с параметром `-show`, на экране появится диалоговое окно с элементами графического пользовательского интерфейса (ГПИ), как показано на рис. 2.7.

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

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

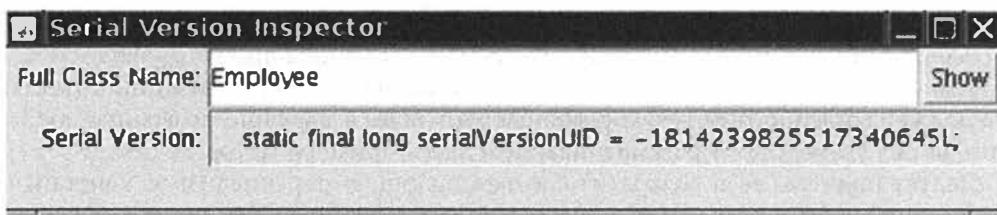


Рис. 2.7. Графическая версия утилиты serialver

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

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

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

Обратимся к конкретному примеру. Допустим, что ряд записей о сотрудниках был сохранен на диске с помощью исходной версии класса `Employee` (версии 1.0), а затем она была заменена версией 2.0 данного класса, в которой введено дополнительное поле данных `department`. На рис. 2.8 показано, что при чтении объекта версии 1.0 в программе, где используются объекты версии 2.0, произойдет следующее: в поле `department` будет установлено пустое значение `null`. А на рис. 2.9 показана обратная ситуация, возникающая при чтении объекта версии 2.0 в программе, где используются объекты версии 1.0. В этом случае дополнительное поле `department` будет просто проигнорировано.

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

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

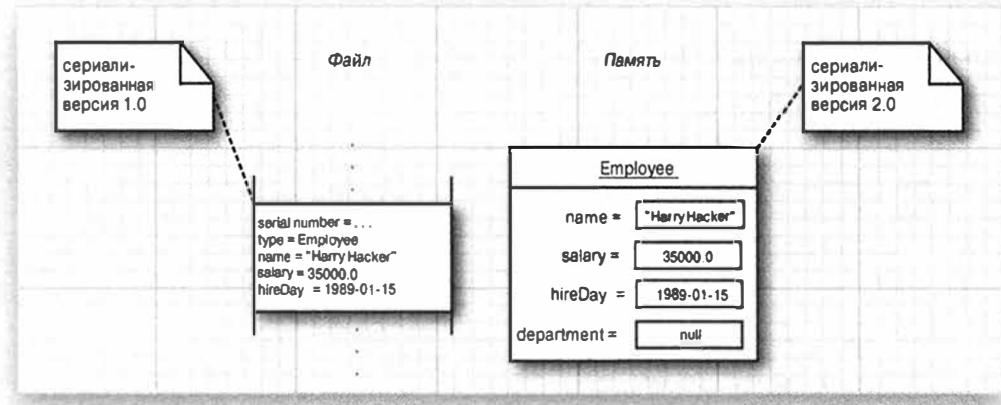


Рис. 2.8. Чтение объекта с меньшим количеством полей данных

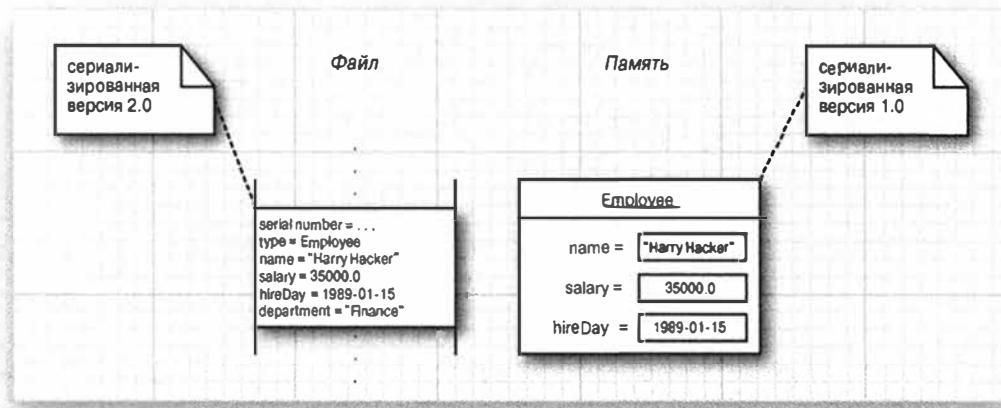


Рис. 2.9. Чтение объекта с большим количеством полей данных

2.4.6. Применение сериализации для клонирования

У механизма сериализации имеется еще один интересный способ применения: он позволяет легко клонировать объект, при условии, что класс последнего является сериализуемым. Для этого достаточно сериализовать объект в поток вывода, а затем прочитать его обратно. В результате получится новый объект, представляющий собой точную (полную) копию того, что уже существует. Записывать этот

объект в файл совсем не обязательно. Вместо этого можно воспользоваться потоком вывода типа `ByteArrayOutputStream`, сохранив данные в байтовом массиве.

Как демонстрируется в примере программы из листинга 2.4, чтобы получить клон объекта, достаточно расширить класс `Serializable`. Однако, несмотря на всю изощренность такого способа, он, как правило, оказывается менее быстродействующим, чем способ клонирования, явным образом создающий новый объект и копирующий или клонирующий поля данных.

Листинг 2.4. Исходный код из файла `serialClone/SerialCloneTest.java`

```
1 package serialClone;
2
3 /**
4  * @version 1.21 13 Jul 2016
5  * @author Cay Horstmann
6  */
7
8 import java.io.*;
9 import java.util.*;
10 import java.time.*;
11
12 public class SerialCloneTest
13 {
14     public static void main(String[] args)
15         throws CloneNotSupportedException
16     {
17         Employee harry =
18             new Employee("Harry Hacker", 35000, 1989, 10, 1);
19         // клонировать объект harry
20         Employee harry2 = (Employee) harry.clone();
21
22         // видоизменить объект harry
23         harry.raiseSalary(10);
24
25         // теперь оригинал и клон объекта harry отличаются
26         System.out.println(harry);
27         System.out.println(harry2);
28     }
29 }
30
31 /**
32  * Класс, в методе клонирования которого
33  * применяется сериализация
34  */
35
36 class Serializable implements Cloneable, Serializable
37 {
38     public Object clone() throws CloneNotSupportedException
39     {
40         try {
41             // сохранить объект в байтовом массиве
42             ByteArrayOutputStream bout =
43                 new ByteArrayOutputStream();
44             try (ObjectOutputStream out =
45                 new ObjectOutputStream(bout))
46             {
47                 out.writeObject(this);
48             }
49             return (Employee) new ByteArrayInputStream(bout.toByteArray());
50         }
51         catch (IOException e)
52         {
53             throw new CloneNotSupportedException(e.getMessage());
54         }
55     }
56 }
```

```
48      }
49
50      // ввести клон объекта из байтового массива
51      try (InputStream bin =
52          new ByteArrayInputStream(bout.toByteArray()))
53      {
54          ObjectInputStream in = new ObjectInputStream(bin);
55          return in.readObject();
56      }
57  }
58  catch (IOException | ClassNotFoundException e)
59  {
60      CloneNotSupportedException e2 =
61          new CloneNotSupportedException();
62      e2.initCause(e);
63      throw e2;
64  }
65 }
66 }
67
68 /**
69 * Класс Employee, переопределяемый для расширения
70 * класса SerialCloneable
71 */
72 class Employee extends SerialCloneable
73 {
74     private String name;
75     private double salary;
76     private LocalDate hireDay;
77
78     public Employee(String n, double s, int year,
79                     int month, int day)
79
80     {
81         name = n;
82         salary = s;
83         hireDay = LocalDate.of(year, month, day);
84     }
85
86     public String getName()
87     {
88         return name;
89     }
90
91     public double getSalary()
92     {
93         return salary;
94     }
95
96     public LocalDate getHireDay()
97     {
98         return hireDay;
99     }
100
101 /**
102 * Поднимает зарплату данному работнику
103 * @byPercent Процент повышения зарплаты
104 */
105 public void raiseSalary(double byPercent)
106 {
```

```
107     double raise = salary * byPercent / 100;
108     salary += raise;
109 }
110
111 public String toString()
112 {
113     return getClass().getName()
114         + "[name=" + name
115         + ",salary=" + salary
116         + ",hireDay=" + hireDay
117         + "]";
118 }
119 }
```

2.5. Манипулирование файлами

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

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

2.5.1. Пути к файлам

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

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

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

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

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



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

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

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

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

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

Имеется сокращенный вариант метода `resolve()`, принимающий в качестве своего параметра символьную строку вместо пути:

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

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

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

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

Метод `normalize()` удаляет любые избыточные знаки `.` и `..` или иные составляющие пути, которые считаются лишними в файловой системе. Например, в результате нормализации пути `/home/harry/.../fred/../input.txt` получается путь `/home/fred/input.txt`. А метод `toAbsolutePath()` порождает

абсолютный путь из заданного пути, начиная с коренной составляющей, например /home/fred/input.txt или c:\Users\fred\input.txt.

В интерфейсе Path имеется немало полезных методов для разделения путей к файлам на составляющие. В приведенном ниже фрагменте кода демонстрируется применение наиболее полезных методов из этого интерфейса.

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

Как пояснялось в первом томе настоящего издания, из объекта типа Path можно создать объект типа Scanner следующим образом:

```
Scanner in = new Scanner(Paths.get("/home/fred/input.txt"));
```



На заметку! Время от времени вам, возможно, придется иметь дело с унаследованными прикладными программными интерфейсами API, использующими класс **File** вместо интерфейса **Path**. Так, в интерфейсе **Path** имеется метод **toFile()**, тогда как в классе **File** — метод **toPath()**.

java.nio.file.Paths 7

- **static Path get(String first, String... more)**
Составляет путь, соединяя заданные символьные строки.

java.nio.file.Path 7

- **Path resolve(Path other)**
- **Path resolve(String other)**
Если параметр **other** содержит абсолютный путь, то возвращается путь **other**, а иначе — путь, получаемый в результате объединения путей **this** и **other**.
- **Path resolveSibling(Path other)**
- **Path resolveSibling(String other)**
Если параметр **other** содержит абсолютный путь, то возвращается путь **other**, а иначе — путь, получаемый в результате объединения родительского пути **this** и заданного пути **other**.
- **Path relativize(Path other)**
Возвращает относительный путь, порождающий путь **other** при разрешении пути **this**.
- **Path normalize()**
Удаляет из пути избыточные составляющие, например, знаки . и ...
- **Path toAbsolutePath()**
Возвращает абсолютный путь, равнозначный данному пути.
- **Path getParent()**
Возвращает родительский путь или пустое значение **null**, если у данного пути отсутствует родительский путь.

java.nio.file.Path 7 (окончание)**• Path getFileName()**

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

• Path getRoot()

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

• toFile()

Составляет объект типа `File` из данного пути.

java.io.File 1.0**• Path toPath() 7**

Составляет объект типа `Path` из данного пути.

2.5.2. Чтение и запись данных в файлы

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

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

Если же требуется прочитать содержимое файла в виде символьной строки, сначала необходимо вызвать метод `readAllBytes()`, как показано выше, а затем следующий конструктор:

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

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

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

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

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

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

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

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

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

java.nio.file.Files 7

- **static byte[] readAllBytes(Path path)**
- **static List<String> readAllLines(Path path, Charset charset)**
Читают содержимое файла.
- **static Path write(Path path, byte[] contents, OpenOption... options)**
- **static Path write(Path path, Iterable<? extends CharSequence> contents, OpenOption options)**
Записывают заданное содержимое в файл и возвращают **path** как путь к нему.
- **static InputStream newInputStream(Path path, OpenOption... options)**
- **static OutputStream newOutputStream(Path path, OpenOption... options)**
- **static BufferedReader newBufferedReader(Path path, Charset charset)**
- **static BufferedWriter newBufferedWriter(Path path, Charset charset, OpenOption... options)**
Открывают файл для чтения или записи.

2.5.3. Создание файлов и каталогов

Чтобы создать новый каталог, достаточно сделать следующий вызов:

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

Все составляющие пути к каталогу, кроме последней, должны уже существовать. А для создания промежуточных каталогов достаточно сделать такой вызов:

```
Files.createDirectories(path);
```

Чтобы создать пустой файл, следует сделать приведенный ниже вызов.

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

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

Для создания временного файла или каталога в указанном месте файловой системы имеются удобные методы. Примеры их применения демонстрируются в приведенном ниже фрагменте кода, где **dir** — это объект типа **Path**, а **prefix**/**suffix** — символьные строки, которые могут быть нулевыми (**null**). Например, в результате вызова **Files.createTempFile(null, ".txt")** может быть возвращен путь **/tmp/1234405522364837194.txt**.

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

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

java.nio.file.Files 7

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

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

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

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

2.5.4. Копирование, перемещение и удаление файлов

Чтобы скопировать файл из одного места в другое, достаточно сделать следующий вызов:

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

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

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

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

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

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

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

В файл, находящийся по пути, указанному в объекте типа Path, можно также скопировать поток ввода. Это, по существу, означает сохранение потока ввода на диске. Аналогично содержимое файла, находящегося по пути, указанному в объекте типа Path, можно направить в поток вывода. В приведенном ниже фрагменте кода показано, каким образом выполняются обе эти операции. Как и в других вызовах метода copy(), по мере надобности можно предоставить дополнительные параметры.

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

И наконец, для удаления файла достаточно вызвать метод
Files.delete(path);

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

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

В табл. 2.3 сведены дополнительные параметры, доступные при выполнении операций с файлами.

Таблица 2.3. Стандартные параметры для операций с файлами

Параметр	Описание
<code>StandardOpenOption</code> ; применяется в потоках ввода-вывода типа <code>newBufferedWriter</code> , <code>newInputStream</code> , <code>newOutputStream</code> для операции записи	
<code>READ</code>	Открыть файл для чтения
<code>WRITE</code>	Открыть файл для записи
<code>APPEND</code>	Если файл открыт для записи, присоединить данные в конце этого файла
<code>TRUNCATE_EXISTING</code>	Если файл открыт для записи, удалить его текущее содержимое
<code>CREATE_NEW</code>	Создать новый файл, указать на неудачный исход операции, если файл уже существует
<code>CREATE</code>	Создать файл атомарно, если файл не существует
<code>DELETE_ON_CLOSE</code>	Удалить файл наилучшим образом после его закрытия
<code>SPARSE</code>	Указать файловой системе, что этот файл окажется разреженным
<code>DSYNC SYNC</code>	Потребовать, чтобы при каждом обновлении файла данные и метаданные синхронно записывались на запоминающем устройстве
<code>StandardCopyOption</code> ; применяется в операциях копирования и перемещения	
<code>ATOMIC_MOVE</code>	Переместить файл атомарно
<code>COPY_ATTRIBUTES</code>	Скопировать атрибуты файла
<code>REPLACE_EXISTING</code>	Заменить целевой файл, если он существует
<code>LinkOption</code> ; применяется во всех упомянутых выше методах, а также в методах <code>exists()</code> , <code>isDirectory()</code> , <code>isRegularFile()</code>	
<code>NOFOLLOW_LINKS</code>	Не следовать по символическим ссылкам
<code>FileVisitOption</code> ; применяется в методах <code>find()</code> , <code>walk()</code> , <code>walkFileTree()</code>	
<code>FOLLOW_LINKS</code>	Следовать по символическим ссылкам

java.nio.file.Files 7

- **static Path copy(Path from, Path to, CopyOption... options)**
- **staticPath move(Path from, Path to, CopyOption... options)**
Копируют или перемещают файл из исходного места *from* в заданное целевое место *to*, возвращая последнее.
- **static long copy(InputStream from, Path to, CopyOption... options)**
- **static long copy(Path from, OutputStream to, CopyOption... options)**
Копируют данные в файл из потока ввода или из файла в поток вывода, возвращая количество скопированных байтов.
- **static void delete(Path path)**
- **static boolean deleteIfExists(Path path)**
Удаляют заданный файл или пустой каталог. Первый метод генерирует исключение, если заданный файл или каталог не существует. А второй метод возвращает в этом случае логическое значение *false*.

2.5.5. Получение сведений о файлах

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

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

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

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

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

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

Для получения перечисленных выше атрибутов файлов достаточно сделать следующий вызов:

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

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

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

А далее можно определить группового или индивидуального владельца файла, а также права на групповой или глобальный доступ к нему. Мы не будем здесь вдаваться в подробности этого процесса, поскольку большая часть сведений о файлах не переносится из одной операционной системы в другую.

`java.nio.file.Files` 7

- `static boolean exists(Path path)`
- `static boolean isHidden(Path path)`
- `static boolean isReadable(Path path)`
- `static boolean isWritable(Path path)`
- `static boolean isExecutable(Path path)`
- `static boolean isRegularFile(Path path)`
- `static boolean isDirectory(Path path)`
- `static boolean isSymbolicLink(Path path)`
Проверяют заданное свойство файла по указанному пути.
- `static long size(Path path)`
Получает размер файла в байтах.
- `A readAttributes(Path path, Class<A> type, LinkOption... options)`
Читает атрибуты файла, относящиеся к типу `A`

`java.nio.file.attribute.BasicFileAttributes` 7

- `FileTime creationTime()`
- `FileTime lastAccessTime()`
- `FileTime lastModifiedTime()`
- `boolean isRegularFile()`
- `boolean isDirectory()`
- `boolean isSymbolicLink()`
- `long size()`
- `Object fileKey()`
Получают запрашиваемый атрибут файла.

2.5.6. Обход элементов каталога

Статический метод `Files.list()` возвращает поток данных типа `Stream<Path>`, откуда читаются элементы каталога. Содержимое каталога читается по требованию, и благодаря этому становится возможной эффективная обработка каталогов с большим количеством элементов.

Для чтения содержимого каталога требуется закрыть системные ресурсы, поэтому данную операцию необходимо заключить в блок оператора `try` следующим образом:

```
try (Stream<Path> entries = Files.list(pathToDirectory)) {  
    ...  
}
```

Метод `list()` не входит в подкаталоги. Чтобы обработать все порожденные элементы каталога, следует воспользоваться методом `Files.walk()`:

```
try (Stream<Path> entries = Files.walk(pathToRoot)) {  
    // Содержит все порожденные элементы, обойденные в глубину  
}
```

Ниже приведен пример обхода дерева каталогов из разархивированного файла `src.zip`. Как видите, всякий раз, когда в результате обхода получается каталог, происходит вход в него, прежде чем продолжать обход родственных ему каталогов.

```
java  
java/nio  
java/nio/DirectCharBufferU.java  
java/nio/ByteBufferAsShortBufferRL.java  
java/nio/MappedByteBuffer.java  
...  
java/nio/ByteBufferAsDoubleBufferB.java  
java/nio/charset  
java/nio/charset/CoderMalfunctionError.java  
java/nio/charset/CharsetDecoder.java  
java/nio/charset/UnsupportedCharsetException.java  
java/nio/charset/spi  
java/nio/charset/spi/CharsetProvider.java  
java/nio/charset/StandardCharsets.java  
java/nio/charset/Charset.java  
...  
java/nio/charset/CoderResult.java  
java/nio/HeapFloatBufferR.java  
...
```

Глубину дерева каталогов, которое требуется обойти, можно ограничить, вызвав метод `Files.walk(pathToRoot, depth)`. В обоих рассматриваемых здесь вызовах метода `walk()` имеются аргументы переменной длины типа `FileVisitOption...`, но для перехода по символическим ссылкам можно предоставить только параметр `FOLLOW_LINKS`.



На заметку! Чтобы отфильтровать пути, возвращаемые методом `walk()` по критерию, включающему в себя атрибуты файлов, хранящиеся в каталоге, в том числе размер, время создания или тип (файла, каталога, символьской ссылки), вместо метода `walk()` лучше воспользоваться методом `find()`. Этот метод следует вызывать с предикатной функцией, принимающей в качестве параметров путь и объект типа `BasicFileAttributes`. Единственное преимущество такого подхода состоит в его эффективности. Ведь чтение каталога происходит в любом случае, и поэтому атрибуты сразу же становятся доступными.

В следующем фрагменте кода метод `Files.walk()` применяется для копирования одного каталога в другой:

```
Files.walk(source).forEach(p ->
{
    try {
        Path q = target.resolve(source.relativize(p));
        if (Files.isDirectory(p))
            Files.createDirectory(q);
        else
            Files.copy(p, q);
    } catch (IOException ex) {
        throw new UncheckedIOException(ex);
    }
});
```

К сожалению, методом `Files.walk()` не так-то просто воспользоваться для удаления дерева каталогов, поскольку для этого нужно обойти все порожденные элементы, прежде чем удалить родительский элемент. В следующем разделе поясняется, как преодолеть подобное затруднение.

2.5.7. Применение потоков каталогов

Как пояснялось в предыдущем разделе, метод `Files.walk()` возвращает поток данных типа `Stream<Path>` для обхода порожденных элементов каталога. Но иногда требуется более точный контроль над процессом обхода элементов каталога. И в таком случае следует воспользоваться потоком данных типа `Files.newDirectoryStream`, который производит поток данных типа `DirectoryStream`. Следует, однако, иметь в виду, что он не относится к интерфейсу, подчиненному интерфейсу `java.util.stream.Stream`, специально предназначенному для обхода элементов каталога. Напротив, он относится к интерфейсу, подчиненному интерфейсу `Iterable`, что дает возможность использовать поток каталогов в расширенном цикле `for`, как демонстрируется в следующем примере кода:

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

Блок оператора `try` с ресурсами обеспечивает надлежащее закрытие потока ввода из каталога. Определенного порядка обхода элементов каталога не существует. Файлы можно отфильтровать по глобальному шаблону, как показано ниже. Все глобальные шаблоны перечислены в табл. 2.4.

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

Таблица 2.4. Глобальные шаблоны

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



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

Если требуется обойти порожденные элементы каталога, следует вызвать метод `walkFileTree()` с объектом типа `FileVisitor` в качестве параметра. Этот объект уведомляется в следующих случаях.

- Когда встречается файл или каталог: `FileVisitResult visitFile(T path, BasicFileAttributes attrs)`.
- До обработки каталога: `FileVisitResult preVisitDirectory(T dir, IOException ex)`.
- После обработки каталога: `FileVisitResult postVisitDirectory(T dir, IOException ex)`.
- Когда возникает ошибка в связи с попыткой обратиться к файлу или каталогу, например, открыть каталог без надлежащих прав доступа: `FileVisitResult visitFileFailed(T path, IOException ex)`.

В каждом случае можно указать следующее.

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

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



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

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

```
Files.walkFileTree(Paths.get("/"), new SimpleFileVisitor<Path>()
{
    public FileVisitResult preVisitDirectory(Path path,
                                              BasicFileAttributes attrs) throws IOException
    {
        System.out.println(path);
        return FileVisitResult.CONTINUE;
    }
    public FileVisitResult postVisitDirectory(Path dir,
                                              IOException exc)
    {
        return FileVisitResult.CONTINUE;
    }
    public FileVisitResult visitFileFailed(Path path,
                                          IOException exc) throws IOException
    {
        return FileVisitResult.SKIP_SUBTREE;
    }
});
```

Однако методы `postVisitDirectory()` и `visitFileFailed()` придется переопределить, иначе обращение к файлам сразу же завершится аварийно, как только встретится каталог, который не разрешается открывать. Следует также иметь в виду, что атрибуты пуги передаются методам `postVisitDirectory()` и `visitFile()` в качестве параметра. При обходе каталога уже пришлось обратиться к операционной системе для получения атрибутов, поскольку нужно каким-то образом отличать файлы от каталогов. Благодаря этому исключается необходимость делать еще один вызов.

Остальные методы из интерфейса `FileVisitor` полезны в том случае, если требуется выполнить служебные операции для входа и выхода из каталога. Например, при удалении дерева каталогов после всех файлов необходимо удалить и каталог, в котором они находились. Ниже приведен полный пример кода для удаления дерева каталогов.

```
// удалить дерево каталогов, начиная с корневого каталога
Files.walkFileTree(root, new SimpleFileVisitor<Path>()
{
    public FileVisitResult visitFile(Path file,
                                    BasicFileAttributes attrs) throws IOException
```

```

{
    Files.delete(file);
    return FileVisitResult.CONTINUE;
}
public FileVisitResult postVisitDirectory(Path dir,
                                         IOException e) throws IOException
{
    if (e != null) throw e;
    Files.delete(dir);
    return FileVisitResult.CONTINUE;
}
});

```

java.nio.file.Files 7

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

Получают итератор для обхода всех файлов и каталогов в данном каталоге. Второй метод принимает только те элементы файловой системы, которые совпадают с заданным глобальным шаблоном.

- static Path walkFileTree(Path start, FileVisitor<? super Path> visitor)

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

java.nio.file.SimpleFileVisitor<T> 7

- static FileVisitResult visitFile(T path, BasicFileAttributes attrs)

Вызывается при обращении к файлу или каталогу. Возвращает одно из значений констант **CONTINUE**, **SKIP_SUBTREE**, **SKIP_SIBLINGS** или **TERMINATE**. В стандартной реализации по умолчанию ничего особенного не делает и только продолжает выполнение.

- static FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)

- static FileVisitResult postVisitDirectory(T dir, BasicFileAttributes attrs)

Вызываются до и после обращения к каталогу. В исходной реализации по умолчанию ничего особенного не делают и только продолжают выполнение.

- static FileVisitResult visitFileFailed(T path, IOException exc)

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

2.5.8. Системы ZIP-файлов

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

в результате следующего вызова устанавливается файловая система, содержащая все файлы в ZIP-архиве:

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

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

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

Здесь метод `fs.getPath()` выполняет функции, аналогичные методу `Paths.get()`, для произвольной файловой системы. Чтобы перечислить все файлы в ZIP-архиве, следует обойти дерево файлов, как показано в приведенном ниже фрагменте кода. Это намного удобнее, чем пользоваться прикладным программным интерфейсом API, описанным в разделе 2.3.3, где требовался новый ряд классов только для обращения с ZIP-архивами.

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

`java.nio.file.FileSystems` 7

- **`static FileSystem newFileSystem(Path path, ClassLoader loader)`**

Обходит все установленные поставщики файловых систем, а также файловые системы, которые способен загрузить указанный загрузчик классов, если не указано пустое значение `null` параметра `loader`. По умолчанию предоставается поставщик для систем ZIP-файлов, принимающий файлы с расширением `.zip` или `.jar`.

`java.nio.file.FileSystem` 7

- **`static Path getPath(String first, String... more)`**

Составляет путь, объединяя заданные символьные строки.

2.6. Файлы, отображаемые в памяти

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

2.6.1. Эффективность файлов, отображаемых в памяти

В конце этого раздела приведен пример программы, вычисляющей контрольную сумму CRC32 для файла с использованием операции ввода данных из традиционного и отображаемого в памяти файла. В табл. 2.5 перечислены временные характеристики, полученные на одном компьютере при вычислении с помощью этой программы контрольной суммы для файла `rt.jar` объемом 37 Мбайт, входящего в состав JDK и расположенного в каталоге `jre/lib`.

Таблица 2.5. Временные характеристики некоторых операций над файлами

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

Как следует из табл. 2.5, на отдельно взятой машине при отображении файла в памяти потребовалось немного меньше времени, чем при последовательном вводе данных из файла с буферизацией, и значительно меньше времени, чем при произвольном доступе к файлу средствами класса `RandomAccessFile`. Разумеется, на других машинах эти показатели будут выглядеть несколько иначе, но не вызывает никаких сомнений, что выигрыш в производительности может оказаться значительным при отображении файла в памяти по сравнению с произвольным доступом к нему. В то же время для последовательного ввода из файлов среднего размера прибегать к отображению в памяти нецелесообразно.

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

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

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

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

- `FileChannel.MapMode.PRIVATE`. Получаемый в итоге буфер служит как для чтения, так и для записи, но любые вносимые изменения относятся только к этому буферу, а следовательно, они не будут распространяться на файл.

Получив требуемый буфер, можно перейти непосредственно к чтению и записи данных с помощью методов из класса `ByteBuffer` и его суперкласса `Buffer`. Буфера поддерживают как последовательный, так и произвольный доступ к данным. В любом буфере имеется *позиция*, продвигаемая методами `get()` и `put()`. В качестве примера ниже приведен код, требующийся для последовательного обхода всех байтов в буфере.

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

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

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

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

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

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

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

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

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

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

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



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

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

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

В листинге 2.5 приведен пример программы, вычисляющей для файла контрольную сумму в 32-разрядном циклическом коде с избыточностью (CRC32). Такая контрольная сумма часто применяется для того, чтобы выяснить, не поврежден ли файл. Повреждение файла практически неизбежно ведет к изменению этой контрольной суммы. В состав пакета `java.util.zip` входит класс `CRC32`, позволяющий вычислять такую контрольную сумму для последовательности байтов в следующем цикле:

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



На заметку! Подробное описание алгоритма CRC имеется по адресу <http://www.relishsoft.com/Science/CrcMath.html>.

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

```
java memoryMap.MemoryMapTest имя_файла
```

Листинг 2.5. Исходный код из файла `memoryMap/MemoryMapTest.java`

```
1 package memoryMap;
2
3 import java.io.*;
4 import java.nio.*;
5 import java.nio.channels.*;
6 import java.nio.file.*;
7 import java.util.zip.*;
8 /**
9  * В этой программе контрольная сумма CRC32
10 * вычисляется для файла четырьмя способами
11 * Использование: java memoryMap.MemoryMapTest имя_файла
12 * @version 1.01 2012-05-30
13 * @author Cay Horstmann
14 */
15 public class MemoryMapTest
16 {
17     public static long checksumInputStream(Path filename)
18             throws IOException
19     {
```

```
20     try (InputStream in = Files.newInputStream(filename))
21     {
22         CRC32 crc = new CRC32();
23
24         int c;
25         while ((c = in.read()) != -1)
26             crc.update(c);
27         return crc.getValue();
28     }
29 }
30
31 public static long checksumBufferedInputStream(Path filename)
32     throws IOException
33 {
34     try (InputStream in =
35          new BufferedInputStream(Files.newInputStream(filename)))
36     {
37         CRC32 crc = new CRC32();
38
39         int c;
40         while ((c = in.read()) != -1)
41             crc.update(c);
42         return crc.getValue();
43     }
44 }
45
46 public static long checksumRandomAccessFile(Path filename)
47     throws IOException
48 {
49     try (RandomAccessFile file =
50          new RandomAccessFile(filename.toFile(), "r"))
51     {
52         long length = file.length();
53         CRC32 crc = new CRC32();
54
55         for (long p = 0; p < length; p++)
56         {
57             file.seek(p);
58             int c = file.readByte();
59             crc.update(c);
60         }
61         return crc.getValue();
62     }
63 }
64
65 public static long checksumMappedFile(Path filename)
66     throws IOException
67 {
68     try (FileChannel channel = FileChannel.open(filename))
69     {
70         CRC32 crc = new CRC32();
71         int length = (int) channel.size();
72         MappedByteBuffer buffer =
73             channel.map(FileChannel.MapMode.READ_ONLY, 0, length);
74
75         for (int p = 0; p < length; p++)
76         {
77             int c = buffer.get(p);
78             crc.update(c);
79         }
80     }
```

```
80         return crc.getValue();
81     }
82 }
83
84 public static void main(String[] args) throws IOException
85 {
86     System.out.println("Input Stream:");
87     long start = System.currentTimeMillis();
88     Path filename = Paths.get(args[0]);
89     long crcValue = checksumInputStream(filename);
90     long end = System.currentTimeMillis();
91     System.out.println(Long.toHexString(crcValue));
92     System.out.println((end - start) + " milliseconds");
93
94     System.out.println("Buffered Input Stream:");
95     start = System.currentTimeMillis();
96     crcValue = checksumBufferedInputStream(filename);
97     end = System.currentTimeMillis();
98     System.out.println(Long.toHexString(crcValue));
99     System.out.println((end - start) + " milliseconds");
100
101    System.out.println("Random Access File:");
102    start = System.currentTimeMillis();
103    crcValue = checksumRandomAccessFile(filename);
104    end = System.currentTimeMillis();
105    System.out.println(Long.toHexString(crcValue));
106    System.out.println((end - start) + " milliseconds");
107
108    System.out.println("Mapped File:");
109    start = System.currentTimeMillis();
110    crcValue = checksumMappedFile(filename);
111    end = System.currentTimeMillis();
112    System.out.println(Long.toHexString(crcValue));
113    System.out.println((end - start) + " milliseconds");
114 }
115 }
```

java.io.FileInputStream 1.0

- **FileChannel getChannel()** 1.4

Возвращает канал для получения доступа к данному потоку ввода.

java.io.FileOutputStream 1.0

- **FileChannel getChannel()** 1.4

Возвращает канал для получения доступа к данному потоку вывода.

java.io.RandomAccessFile 1.0

- **FileChannel getChannel()** 1.4

Возвращает канал для получения доступа к данному файлу.

java.nio.channels.FileChannel 1.4

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

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

Параметры:

path

Путь к файлу, для доступа к

которому открывается канал

options

Принимает значения следующих

констант из перечисления типа

**StandardOpenOption: WRITE,
APPEND, TRUNCATE_EXISTING, CREATE**

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

Отображает часть файла в памяти.

Параметры:

mode

Принимает значения следующих

констант из класса

**FileChannel.MapMode: READ_ONLY,
READ_WRITE или PRIVATE**

position

Начало отображаемой части файла

size

Размер отображаемой части файла

java.nio.Buffer 1.4

- **boolean hasRemaining()**

Возвращает логическое значение **true**, если текущее положение в буфере еще не достигло предельной позиции.

- **int limit()**

Возвращает предельную позицию в буфере, т.е. первую позицию, где больше нет никаких значений.

java.nio.ByteBuffer 1.4

- **byte get()**

Получает байт из текущей позиции и продвигает текущую позицию к следующему байту.

- **byte get(int index)**

Получает байт по указанному индексу.

- **ByteBuffer put(byte b)**

Размещает байт на текущей позиции и продвигает ее к следующему байту. Возвращает ссылку на данный буфер.

java.nio.ByteBuffer 1.4 (окончание)**• ByteBuffer put(int index, byte b)**

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

• ByteBuffer get(byte[] destination)**• ByteBuffer get(byte[] destination, int offset, int length)**

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

Параметры:

destination

Заполняемый байтовый массив

offset

Смещение заполняемой части массива

length

Длина заполняемой части массива

• ByteBuffer put(byte[] source)**• ByteBuffer put(byte[] source, int offset, int length)**

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

Параметры:

source

Записываемый байтовый массив

offset

Смещение записываемой части массива

length

Длина записываемой части массива

• Xxx getXxx()**• Xxx getXxx(int index)****• ByteBuffer putXxx(Xxx value)****• ByteBuffer putXxx(int index, Xxx value)**

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

• ByteBuffer order(ByteOrder order)**• ByteOrder order()**

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

• static ByteBuffer allocate(int capacity)

Конструирует буфер заданной емкости.

• static ByteBuffer wrap(byte[] values)

Конструирует буфер, опирающийся на заданный массив.

• CharBuffer asCharBuffer()

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

java.nio.CharBuffer 1.4

- `char get()`
 - `CharBuffer get(char[] destination)`
 - `CharBuffer get(char[] destination, int offset, int length)`
- Получают значение типа `char` или ряд подобных значений, начиная с указанной позиции в буфере и продвигая ее на количество считанных символов. Два последних метода возвращают ссылку `this` на данный буфер.
- `CharBuffer put(char c)`
 - `CharBuffer put(char[] source)`
 - `CharBuffer put(char[] source, int offset, int length)`
 - `CharBuffer put(String source)`
 - `CharBuffer put(CharBuffer source)`

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

2.6.2. Структура буфера данных

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

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



На заметку! Класс `StringBuffer` не имеет никакого отношения к этим подклассам, реализующим буфера данных.

На практике чаще всего применяются классы `ByteBuffer` и `CharBuffer`. Как показано на рис. 2.10, каждый буфер обладает следующим свойствами.

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

Все эти свойства удовлетворяют следующему условию:

$$0 = \text{отметка} = \text{позиция} = \text{предел} = \text{емкость}$$

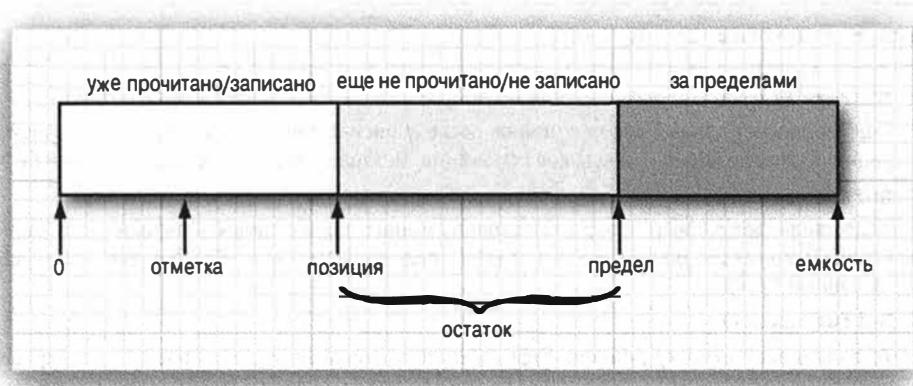


Рис. 2.10. Буфер

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

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

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

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

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

java.nio.Buffer 1.4

- **Buffer clear()**

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

java.nio.Buffer 1.4 (окончание)

- **Buffer flip()**

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

- **Buffer rewind()**

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

- **Buffer mark()**

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

- **Buffer reset()**

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

- **int remaining()**

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

- **int position()**

- **void position(int newValue)**

Получают или устанавливают текущую позицию в данном буфере.

- **int capacity()**

Возвращает емкость данного буфера.

2.6.3. Блокирование файлов

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

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

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

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

или

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

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

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

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

или

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

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



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

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

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

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

- В некоторых системах блокирование файлов является лишь *желательным*, но не *обязательным*. Если прикладной программе не удастся получить разрешение на блокировку, она все равно может начать запись данных файл, несмотря на то, что доступ к этому файлу в данный момент заблокирован другой прикладной программой.
- В некоторых системах нельзя одновременно блокировать файл и отображать его в памяти.

- Блокировки файлов удерживаются всей виртуальной машиной Java. Если с помощью одной и той же виртуальной машины запускаются сразу две программы (например, аплет или программа запуска приложений), то они не смогут по отдельности получать разрешение на блокировку одного и того же файла. А если виртуальная машина уже удерживает другую перекрывающую блокировку для того же самого файла, то методы `lock()` и `tryLock()` будут просто генерировать исключение типа `OverlappingFileLockException`.
- В некоторых системах закрытие канала приводит к снятию с базового файла всех блокировок, которые удерживаются виртуальной машиной Java. Поэтому лучше не открывать много каналов для одного и того же заблокированного файла.
- Блокирование файлов в сетевой файловой системе очень сильно зависит от используемой системы, и поэтому в таких системах лучше всего избегать этого механизма.

java.nio.channels.FileChannel 1.4• **`FileLock lock()`**

Получает разрешение на исключительную блокировку всего файла. Этот метод блокируется до тех пор, пока не получит разрешение на блокировку.

• **`FileLock tryLock()`**

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

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

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

Параметры:

`position`

Начало блокируемой части файла

`size`

Размер блокируемой части файла

`shared`

Принимает логическое значение `true`, если требуется разделяемая блокировка, или логическое значение `false`, если требуется исключительная блокировка

java.nio.channels.FileLock 1.4• **`void close() 1.7`**

Снимает блокировку.

2.7. Регулярные выражения

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

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

- начинается с буквы J или j;
- содержит ava на месте трех последующих букв;
- а в остальной части содержит один или несколько произвольных символов.

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

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

- Класс символов — это набор альтернативных символов, заключенных в квадратные скобки, например: [Jj], [0-9], [A-Za-z] или [^0-9]. Здесь знаком — обозначается диапазон символов (т.е. все символы, значения которых в Юникоде находятся в указанных пределах), а знаком ^ — дополнение (т.е. все символы, кроме указанных).
- Чтобы включить знак — в класс символов, его нужно сделать первым или последним элементом данного класса. А для того чтобы включить знак [, его следует сделать первым элементом. И для того чтобы включить знак ^, его достаточно разместить где угодно, только не в начале класса символов. Экранировать необходимо только знаки [и \.
- Имеется немало предопределенных классов символов вроде \d (для цифр) или \p{Sc} (для знака денежной единицы в Юникоде), как показано в табл. 2.6 и 2.7.
- Большинство символов указываются для совпадения непосредственно в шаблоне, как, например, буквы ava в рассмотренном выше шаблоне.
- Знак . обозначает совпадение с любым символом, кроме символов окончания строки (в зависимости от установленных флагов).
- Знак \ служит для экранирования символов, например, выражение \. обозначает совпадение с точкой, а выражение \\ — совпадение с обратной косой чертой.

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

Таблица 2.6. Синтаксис регулярных выражений

Синтаксис	Описание	Пример
Символы		
<code>с</code> , любой символ, кроме знаков <code>,</code> , <code>*</code> , <code>+</code> , <code>?</code> , <code>{</code> , <code>}</code> , <code>(</code> , <code>)</code> , <code>[</code> , <code>\</code> , <code>^</code> , <code>\$</code>	Символ <code>с</code>	<code>]</code>
<code>.</code>	Любой символ, кроме знаков окончания строки, или же любой символ, если установлен флаг <code>DOTALL</code>	
<code>\x{p}</code>	Кодовая точка Юникода, представленная в шестнадцатеричном коде <code>p</code>	<code>\x{1D546}</code>
<code>\uhhhh, \xhh, \0o, \0oo, \0ooo</code>	Кодовая точка Юникода, представленная в шестнадцатеричном или восьмеричном коде	<code>\uFFFF</code>
<code>\a, \e, \f, \n, \r, \t</code>	Предупреждение (<code>\x{7}</code>), переключение кода (<code>\x{1B}</code>), новая строка (<code>\x{A}</code>), возврат каретки (<code>\x{D}</code>), табуляция (<code>\x{9}</code>)	<code>\n</code>

Продолжение табл. 2.6

Синтаксис	Описание	Пример
<code>\cc</code> , где <code>c</code> — буква в пределах <code>[A-Z]</code> или один из знаков <code>0, [, \,], ^, _?</code>	Управляющий символ, соответствующий обозначению <code>c</code>	<code>\cH</code> — возврат на одну позицию (<code>\x{8}</code>)
<code>\c</code> , где <code>c</code> — любой символ, кроме буквы или цифры в пределах <code>[A-Za-z0-9]</code>	Символ <code>c</code>	<code>\\"</code>
<code>\Q . . . \E</code>	Все, что указано от начала и до конца цитаты	Шаблон <code>\Q(...)\E</code> совпадает с символьной строкой (...)
Классы символов		
<code>[C₁C₂...]</code> , где <code>C_i</code> — символы в пределах <code>c-d</code> или классы символов	Любой символ из последовательности <code>C₁, C₂...</code>	<code>[0-9+-]</code>
<code>[^...]</code>	Дополнение класса символов	<code>[^\d\s]</code>
<code>[... && ...]</code>	Пересечение классов символов	<code>(\p{L} && [^A-Za-z])</code>
<code>\p{ . . . }, \P{ . . . }</code>	Предопределенный класс символов (см. табл. 2.7); его дополнение	Шаблон <code>\p{L}</code> совпадает с буквой в Юникоде, как, впрочем, и шаблон <code>\p{L}</code> , а следовательно, фигурную скобку можно опустить
<code>\d, \D</code>	Цифры (по шаблону <code>[0-9]</code> или <code>\p{Digit}</code> , если установлен флаг <code>UNICODE_CHARACTER_CLASS</code>); их дополнение	Шаблон <code>\d+</code> обозначает последовательность цифр
<code>\w, \W</code>	Словесные символы (по шаблону <code>[a-zA-Z0-9_]</code> или словесные символы в Юникоде, если установлен флаг <code>UNICODE_CHARACTER_CLASS</code>); их дополнение	
<code>\s, \S</code>	Пробелы (по шаблону <code>[\n\r\t\f\x{B}]</code> или <code>\p{IsWhite_Space}</code> , если установлен флаг <code>UNICODE_CHARACTER_CLASS</code>); их дополнение	Шаблон <code>\s*</code> , <code>\s*</code> обозначает запятую, отделяемую с обеих сторон дополнительными пробелами
<code>\h, \v, \H, \V</code>	Горизонтальный и вертикальный пробелы, а также их дополнения	
Последовательности и альтернативы		
<code>Xⁿ</code>	Любая строка из выражения <code>X</code> , после которой следует любая строка из выражения <code>Y</code>	Шаблон <code>[1-9] [0-9]*</code> обозначает положительное число без начального нуля
<code>X Y</code>	Любая строка из выражения <code>X</code> или <code>Y</code>	<code>http ftp</code>
Группирование		
<code>(X)</code>	Фиксация совпадения с выражением <code>X</code>	Шаблон <code>'([^\']*')</code> фиксирует текст в кавычках
<code>\n</code>	<code>n</code> -я группа	Шаблон <code>(([""])*\1</code> совпадает со строкой 'Fred' или "Fred", но не со строкой "Fred"

Окончание табл. 2.6

Синтаксис	Описание	Пример
(? <i>имя</i>) <i>X</i>	Фиксация совпадения с выражением <i>X</i> под заданным именем	Шаблон '(? <i>id</i>)' фиксирует совпадение с выражением под именем <i>id</i>
\k <i>имя</i>	Группа с заданным именем	Шаблон \k <i>id</i> совпадает с группой под именем <i>id</i>
(?: <i>X</i>)	Употребление круглых скобок без фиксации выражения <i>X</i>	В шаблоне (?: <i>http ftp</i>)://(.*) происходит совпадение с группой \1 после знаков ://
(? <i>f₁</i> <i>f₂</i> . . . : <i>X</i>), (? <i>f₁</i> . . . - <i>f_k</i> . . . : <i>X</i>), где <i>f_i</i> находится в пределах [димбнУх]	Совпадение, но без фиксации с выражением <i>X</i> при установленных или сброшенных флагах (после знака -)	Шаблон (?i:jpe?g) обозначает совпадение без учета регистра
Прочее (?) . . .)	См. ниже пояснения к классу Pattern в описании прикладного программного интерфейса API	
Кванторы		
<i>X</i> ?	Необязательное наличие выражения <i>X</i>	Шаблон \+? обозначает наличие необязательного знака "плюс"
<i>X</i> * , <i>X</i> +	Повторение 0, 1 или больше раз выражения <i>X</i>	Шаблон [1-9] [0-9]+ обозначает целое число, большее или равное 10
<i>X</i> { <i>n</i> } <i>X</i> { <i>n</i> ,} <i>X</i> { <i>n,m</i> }	Повторение <i>n</i> раз, как минимум <i>n</i> раз, от <i>n</i> до <i>m</i> раз выражения <i>X</i>	Шаблон [0-7]{1,3} обозначает от одной до трех восьмеричных цифр
<i>Q</i> ?, где <i>Q</i> — кванторное выражение	Принудительный квантор, пытающийся найти самое короткое совпадение, прежде чем искать более длинное	Шаблон .*(<.+?>).* фиксирует самую короткую последовательность, заключенную в угловые скобки
<i>Q</i> +, где <i>Q</i> — кванторное выражение	Положительный квантор, принимающий самое длинное совпадение без отката	Шаблон '[^']*' совпадает со строками, заключенными в одиночные кавычки, но сразу же не совпадает со строками без закрывающей кавычки
Обнаружение границ		
^, \$	Начало и конец ввода (или начало и конец строки в многострочном режиме)	Шаблон ^Java\$ обозначает совпадение с введенными данными или строкой Java
\A, \Z, \z	Начало ввода, конец ввода, абсолютный конец ввода (не изменяется в многострочном режиме)	
\b, \B	Словесная граница, несловесная граница	Шаблон \bJava\b обозначает совпадение со словом Java
\G	Конец предыдущего совпадения	

Таблица 2.7. Имена предопределенных классов символов, применяемых с префиксом \p

Имя класса символов	Описание
<code>posixClass</code>	<code>posixClass</code> — один из классов <code>Lower</code> , <code>Upper</code> , <code>Alpha</code> , <code>Digit</code> , <code>Alnum</code> , <code>Punct</code> , <code>Graph</code> , <code>Print</code> , <code>Cntrl</code> , <code>Xdigit</code> , <code>Space</code> , <code>Blank</code> , <code>ASCII</code> , интерпретируемых как класс по стандарту POSIX или Unicode в зависимости от состояния флага <code>UNICODE_CHARACTER_CLASS</code>
<code>IsScript, sc=Script, script=Script</code>	Сценарий, принимаемый методом <code>Character.UnicodeScript.forName()</code>
<code>InBlock, blk=Block, block=Block</code>	Блок, принимаемый методом <code>Character.UnicodeScript.forName()</code>
<code>Category, InCategory, gc=Category, general_category=Category</code>	Одно- или двухбуквенное наименование общей категории символов в Юникоде
<code>IsProperty</code>	<code>Property</code> — одно из имен свойств <code>Alphabetic</code> , <code>Ideographic</code> , <code>Letter</code> , <code>Lowercase</code> , <code>Uppercase</code> , <code>Titlecase</code> , <code>Punctuation</code> , <code>Control</code> , <code>White_Space</code> , <code>Digit</code> , <code>Hex_Digit</code> , <code>Noncharacter_Code_Point</code> , <code>Assigned</code>
<code>javaMethod</code>	Вызов метода <code>Character.isMethod()</code> , который не должен считаться не рекомендованным к применению

В качестве примера ниже приведено непростое, но потенциально полезное регулярное выражение (в нем описываются десятичные или шестнадцатеричные целые числа).

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

К сожалению, синтаксис регулярных выражений не полностью стандартизирован и может выглядеть по-разному в различных программах и библиотеках, где они применяются. И хотя существует общее согласие по базовым конструкциям, тем не менее, имеется масса досадных отличий в деталях. Так, в классах, реализующих регулярные выражения в Java, применяется синтаксис регулярных выражений, подобный, но все-таки не полностью совпадающий с тем, что применяется в языке Perl. В табл. 2.6 перечислены все конструкции этого синтаксиса, внедренного в Java. Более подробные сведения о синтаксисе регулярных выражения можно найти в документации на прикладной программный интерфейс API для класса `Pattern` или в книге *Mastering Regular Expression, 3d Edition* Джеки Фридла (Jeffrey E. F. Friedl; издательство O'Reilly and Associates, 2006 г.; в русском переводе книга вышла под названием *Регулярные выражения*, 3-е издание в издательстве "Символ-Плюс", 2008 г.).

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

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

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

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

А с другой стороны, флаги можно указать в самом шаблоне следующим образом:

```
String regex = "(?iU:выражение)";
```

Флаги, применяемые в регулярных выражениях, перечислены ниже.

- Pattern.CASE_INSENSITIVE или i. Обозначает сопоставление символов с шаблоном без учета регистра. По умолчанию этот флаг принимает во внимание только символы в коде US ASCII.
- Pattern.UNICODE_CASE или u. В сочетании с флагом CASE_INSENSITIVE обозначает сопоставление символов с шаблоном, учитывая регистр букв в Юникоде.
- Pattern.UNICODE_CHARACTER_CLASS или U. Обозначает выбор классов символов по стандарту Unicode, а не POSIX. Подразумевает установку флага UNICODE_CASE.
- Pattern.MULTILINE или m. Обозначает применение знаков ^ и \$ для указания на сопоставление символов с шаблоном в начале и в конце строки, а не во всех входных данных.
- Pattern.UNIX_LINES или d. Обозначает распознавание только '\n' в качестве символа конца строки при сопоставлении символов с шаблонами ^ и \$ в многострочном режиме.
- Pattern.DOTALL или s. Обозначает совпадение со знаком . всех символов, включая и символы конца строки.
- Pattern.COMMENTS или x. Обозначает, что пробелы или комментарии (от знака # и до конца строки) игнорируются.
- Pattern.LITERAL. Обозначает, что шаблон воспринимается буквально и совпадение с ним должно быть точным, за исключением, возможно, регистра букв.
- Pattern.CANON_EQ. Обозначает необходимость учитывать каноническую эквивалентность символов Юникода. Например, символ u, после которого следует знак " (диакритический знак над гласной), будет соответствовать символу ў.

Два последних флага нельзя указывать в самом регулярном выражении. Если требуется найти совпадение с элементам коллекции или потока данных, соответствующий шаблон придется превратить в предикат, как показано в приведенном

ниже фрагменте кода, где переменная `result` содержит все символьные строки, совпадающие с регулярным выражением.

```
Stream<String> strings = . . .;
Stream<String> result = strings.filter(pattern.asPredicate());
```

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

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

Чтобы извлечь совпавшую символьную строку, достаточно сделать следующий вызов:

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

Под нулевой группой подразумеваются все входные данные, а индекс первой фактической группы равен 1. Для получения сведений об общем количестве групп следует вызвать метод `groupCount()`, а для именованных групп служат перечисленные ниже методы.

```
int start(String groupName)
int end(String groupName)
String group(String groupName)
```

Вложенные группы упорядочиваются с помощью круглых скобок. Так, если используется шаблон `((1?[0-9]):([0-5][0-9]))[ap]m` и входные данные `11:59am`, то сопоставитель с шаблоном сообщит о перечисленных ниже группах.

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

В листинге 2.6 приведен исходный код примера программы, где сначала предлагается указать шаблон, а затем — сопоставляемые с ним символьные строки, после чего сообщается, совпадают ли введенные строки с шаблоном. Если же в совпавших введенных строках и шаблоне присутствуют группы, то выводятся границы групп в круглых скобках, как показано в следующем примере:

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

Листинг 2.6. Исходный код из файла regex/RegexTest.java

```
1 package regex;
2
3 import java.util.*;
4 import java.util.regex.*;
5
6 /**
7  * В этой программе производится проверка на совпадение
8  * с регулярным выражением. Для этого следует ввести шаблон
9  * и сопоставляемые с ним символьные строки, а для выхода из
10 * программы – нажать клавишу пробела. Если шаблон содержит
11 * группы, их границы отображаются при совпадении
```

```

12  * @version 1.02 2012-06-02
13  * @author Cay Horstmann
14  */
15 public class RegexTest
16 {
17     public static void main(String[] args)
18         throws PatternSyntaxException
19     {
20         Scanner in = new Scanner(System.in);
21         System.out.println("Enter pattern: ");
22         String patternString = in.nextLine();
23
24         Pattern pattern = Pattern.compile(patternString);
25
26         while (true)
27     {
28             System.out.println("Enter string to match: ");
29             String input = in.nextLine();
30             if (input == null || input.equals("")) return;
31             Matcher matcher = pattern.matcher(input);
32             if (matcher.matches())
33             {
34                 System.out.println("Match");
35                 int g = matcher.groupCount();
36                 if (g > 0)
37                 {
38                     for (int i = 0; i < input.length(); i++)
39                 {
40                     // вывести любые пустые группы
41                     for (int j = 1; j <= g; j++)
42                         if (i == matcher.start(j) && i == matcher.end(j))
43                             System.out.print("(");
44                         // вывести знак ( в начале непустых групп
45                         for (int j = 1; j <= g; j++)
46                             if (i == matcher.start(j)
47                                 && i != matcher.end(j))
48                             System.out.print('\'');
49                         System.out.print(input.charAt(i));
50                         // вывести знак ) в конце непустых групп
51                         for (int j = 1; j <= g; j++)
52                             if (i + 1 != matcher.start(j)
53                                 && i + 1 == matcher.end(j))
54                             System.out.print(')');
55                 }
56                 System.out.println();
57             }
58         }
59         else
60             System.out.println("No match");
61     }
62 }
63 }
```

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

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

В листинге 2.7 приведен пример программы, где этот механизм приводится в действие. В ней отыскиваются и выводятся на экран все гипертекстовые ссылки, имеющиеся на веб-странице. Чтобы запустить эту программу на выполнение, в командной строке нужно указать какой-нибудь веб-адрес, например, следующим образом:

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

Листинг 2.7. Исходный код из файла match/HrefMatch.java

```
1 package match;
2
3 import java.io.*;
4 import java.net.*;
5 import java.nio.charset.*;
6 import java.util.regex.*;
7
8 /**
9  * В этой программе отображаются все веб-адреса на веб-странице
10 * путем сопоставления с регулярным выражением, описывающим
11 * дескриптор <a href=...> разметки в коде HTML.
12 * Для запуска программы следует ввести:
13 * java match.HrefMatch веб-адрес
14 * @version 1.02 2016-07-14
15 * @author Cay Horstmann
16 */
17 public class HrefMatch
18 {
19     public static void main(String[] args)
20     {
21         try
22         {
23             // извлечь символьную строку с веб-адресом (URL)
24             // из командной строки или использовать выбираемый
25             // по умолчанию URL
26             String urlString;
27             if (args.length > 0) urlString = args[0];
28             else urlString = "http://java.sun.com";
29
30             // открыть поток ввода для чтения URL
31             InputStreamReader in = new InputStreamReader(
32                 new URL(urlString).openStream(),
33                 StandardCharsets.UTF_8);
34
35             // прочитать содержимое в построитель символьных строк
36             StringBuilder input = new StringBuilder();
37             int ch;
38             while ((ch = in.read()) != -1)
39                 input.append((char) ch);
40         }
41     }
42 }
```

```

40      // найти все совпадения с шаблоном
41      String patternString =
42          "<a\\s+href\\s*=\\s*(\"[^\"]*\"|[^\\s>]*)\\s*>";
43      Pattern pattern = Pattern.compile(
44          patternString, Pattern.CASE_INSENSITIVE);
45      Matcher matcher = pattern.matcher(input);
46
47      while (matcher.find())
48      {
49          String match = matcher.group();
50          System.out.println(match);
51      }
52  }
53  catch (IOException | PatternSyntaxException e)
54  {
55      e.printStackTrace();
56  }
57 }
58 }
```

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

```

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

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

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

```

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

Если имеется много лексем, их можно извлечь по требованию, как показано ниже.

```
Stream<String> tokens = commas.splitAsStream(input);
```

А если предварительная компиляция шаблона или извлечение лексем по требованию не имеет особого значения, то достаточно вызвать метод `String.split()` следующим образом:

```
String[] tokens = input.split("\\s*,\\s*");
```

java.util.regex.Pattern 1.4

- **static Pattern compile(String expression)**
 - **static Pattern compile(String expression, int flags)**
Компилируют символьную строку с регулярным выражением в объект шаблона для быстрой обработки совпадений.
- Параметры: **expression** Регулярное выражение
flags Один или несколько следующих флагов: **CASE_INSENSITIVE**, **UNICODE_CASE**, **MULTILINE**, **UNIX_LINES**, **DOTALL** или **CANON_EQ**
- **Matcher matcher(CharSequence input)**
 - Возвращает объект типа **Matcher**, который можно использовать для обнаружения совпадений с шаблоном во входных данных.
 - **String[] split(CharSequence input)**
 - **String[] split(CharSequence input, int limit)**
 - **Stream<String> splitAsStream(CharSequence input) 8**

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

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

java.util.regex.Matcher 1.4

- **boolean matches()**
Возвращает логическое значение **true**, если входные данные совпадают с шаблоном.
- **boolean lookingAt()**
Возвращает логическое значение **true**, если с шаблоном совпадает начало входных данных.
- **boolean find()**
- **boolean find(int start)**
Пытаются отыскать следующее совпадение, и если это удается, то возвращают логическое значение **true**.

Параметры: **start** Индекс, с которого начинается поиск

java.util.regex.Matcher 1.4 (окончание)• **int start()**• **int end()**

Возвращают начальную или следующую после конечной позицию текущего совпадения.

• **String group()**

Возвращает текущее совпадение.

• **int groupCount()**

Возвращает сведения о количестве групп во входном шаблоне.

• **int start(int groupIndex)**• **int end(int groupIndex)**

Возвращают начальную или конечную позицию данной группы в текущем совпадении.

Параметры: **groupIndex**

Индекс группы (начиная с 1)

или 0 для обозначения
полного совпадения

• **String group(int groupIndex)**

Возвращает символьную строку, совпадающую с заданной группой.

Параметры: **groupIndex**

Индекс группы (начиная с 1)

или 0 для обозначения
полного совпадения

• **String replaceAll(String replacement)**• **String replaceFirst(String replacement)**

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

Параметры: **replacement**

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

• **static String quoteReplacement(String str) 5.0**

Заключает в кавычки все знаки \ и \$ в символьной строке str.

• **Matcher reset()**• **Matcher reset(CharSequence input)**

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

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

3

ГЛАВА

XML

В этой главе...

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

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

В этой главе описаны основы языка XML, а также инструментальные сред-
ства для обработки данных в формате XML, входящие в состав библиотеки Java.
Как и прежде, здесь рассматриваются случаи, когда применение XML считается

совершенно обоснованным, а также ситуации, в которых можно вполне обойтись и без этого языка, используя другие проверенные временем методики проектирования и программирования.

3.1. Введение в XML

В главе 13 первого тома настоящего издания уже приводились примеры использования *файлов свойств*, описывающих конфигурацию программы. Файл свойств содержит конфигурационные параметры в виде пар, состоящих из имени и значения:

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

Единственный метод для чтения такого файла — использование класса *Properties*. Но это отличное в целом средство не всегда подходит, поскольку во многих случаях формат файла свойств непригоден для описания данных, имеющих сложную структуру. Рассмотрим записи *fontname* и *fontsize* из приведенного выше примера. Их значения было бы удобнее объединить в одном приведенном ниже параметре, поскольку это в большей степени соответствовало бы объектно-ориентированному подходу.

```
font=Times Roman 12
```

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

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

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

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

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

```
<configuration>  
  <title>  
    <font>  
      <name>Helvetica</name>
```

```
<size>36</size>
</font>
</title>
<body>
<font>
    <name>Times Roman</name>
    <size>12</size>
</font>
</body>
<window>
    <width>400</width>
    <height>200</height>
</window>
<color>
    <red>0</red>
    <green>50</green>
    <blue>100</blue>
</color>
<menu>
    <item>Times Roman</item>
    <item>Helvetica</item>
    <item>Goudy Old Style</item>
</menu>
</configuration>
```

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

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



На заметку! Очень удачно составленное описание стандарта XML можно найти по адресу www.xml.com/xml/xml.html.

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

- В отличие от HTML, в XML учитывается регистр символов, поэтому дескрипторы `<h1>` и `<h1>` в XML считаются разными.
- В HTML некоторые закрывающие дескрипторы могут отсутствовать. Например, составитель HTML-документа может пропустить дескриптор `</`

`p>` или ``, если из контекста ясно, где заканчивается абзац или пункт списка. А в XML это не разрешается.

- Для элементов разметки без тела в XML предусмотрена сокращенная запись открывающего дескриптора, совмещенного с закрывающим. В этом случае открывающий дескриптор заканчивается знаком `/`, например ``. Это означает, что наличие закрывающего дескриптора `` подразумевается по умолчанию.
- В XML значения атрибутов должны быть заключены в кавычки, а в HTML кавычки могут отсутствовать. Например, дескриптор `<applet code="MyApplet.class" width=300 height=300>` можно использовать в HTML, но нельзя в XML, где значения атрибутов `width` и `height` нужно обязательно заключить в кавычки следующим образом: `width="300"` и `height="300"`.
- В HTML допускается указывать имена атрибутов без их значений, например `<input type="radio" name="language" value="Java" checked>`. А в XML все атрибуты должны быть указаны со своими значениями, например `checked="true"` или `checked="checked"`.

3.1.1. Структура XML-документа

XML-документ должен начинаться с одного из следующих заголовков:

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

или

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



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

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

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

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

И наконец, тело XML-документа содержит *корневой элемент*, который может состоять из других элементов:

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

```
</title>
. .
</configuration>
```

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



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

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

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

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

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

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

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

чем так, как показано ниже.

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

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

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

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

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

Широко распространенное эмпирическое правило гласит: атрибуты следует использовать не для указания значений, а только при изменении их

интерпретации. Если же непонятно, обозначает ли какой-нибудь атрибут изменение интерпретации значения или нет, то лучше отказаться от атрибута и воспользоваться элементом разметки. Во многих полезных XML-документах атрибуты вообще не употребляются.



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

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

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

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

- *Ссылки на символы* в виде &#десятичное_значение; или &#хшестнадцатеричное_значение;. Например, ссылка №233; или №xD9; обозначает символ é.
- *Ссылки на сущности* в виде &имя;. Так, ссылки на сущности <, >, &, ", ' имеют предопределенные значения и соответствуют знакам <, >, & , " и '. В DTD можно также указать другие ссылки на сущности.
- *Разделы CDATA*, разграничиваемые комбинациями символов <! [CDATA[и]]>. Они предназначены для включения строк со знаками <, > или &, которые не следует интерпретировать как символы разметки, как в приведенном ниже примере.

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

В разделах CDATA не допускается использование символьных строк вроде "]]>", поэтому их следует употреблять очень осторожно. Зачастую они выполняют функции своего рода "лазейки" для внедрения в XML-документ данных в устаревшем формате.

- *Инструкции обработки* — это инструкции для прикладных программ, обрабатывающих XML-документы. Такие инструкции разграничиваются символами <? и ?>, как в приведенном ниже примере.

```
<?xmlstylesheet href="mystyle.css" type="text/css"?>
```

Каждый XML-документ начинается со следующей инструкции обработки:

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

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

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

3.2. Синтаксический анализ XML-документов

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

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

DOM-анализатор проще в употреблении, поэтому сначала рассматривается именно он. А потоковый анализатор обычно применяется для обработки длинных документов, когда для древовидного представления XML-данных требуется большой объем памяти. Кроме того, его можно употреблять для извлечения отдельных элементов XML-документа без учета контекста. Подробнее об этом — далее, в разделе 3.6.

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

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

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

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

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

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

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

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

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



На заметку! Если в качестве источника данных служит произвольный поток ввода, синтаксический анализатор не сможет найти те файлы, расположение которых указано относительно данного документа, например, DTD-файл, находящийся в том же каталоге. Для преодоления этого препятствия достаточно установить так называемый “определитель сущностей”. Подробнее об этом можно узнать по адресу www.xml.com/pub/a/2004/03/03/catalogs.html или www.ibm.com/developerworks/xml/library/x-wxid3.html.

Объект типа Document является внутренним представлением древовидной структуры XML-документа. Он состоит из экземпляров классов, реализующих интерфейс Node (Узел) и различные, производные от него интерфейсы. На рис. 3.1 показана иерархия наследования интерфейса Node.

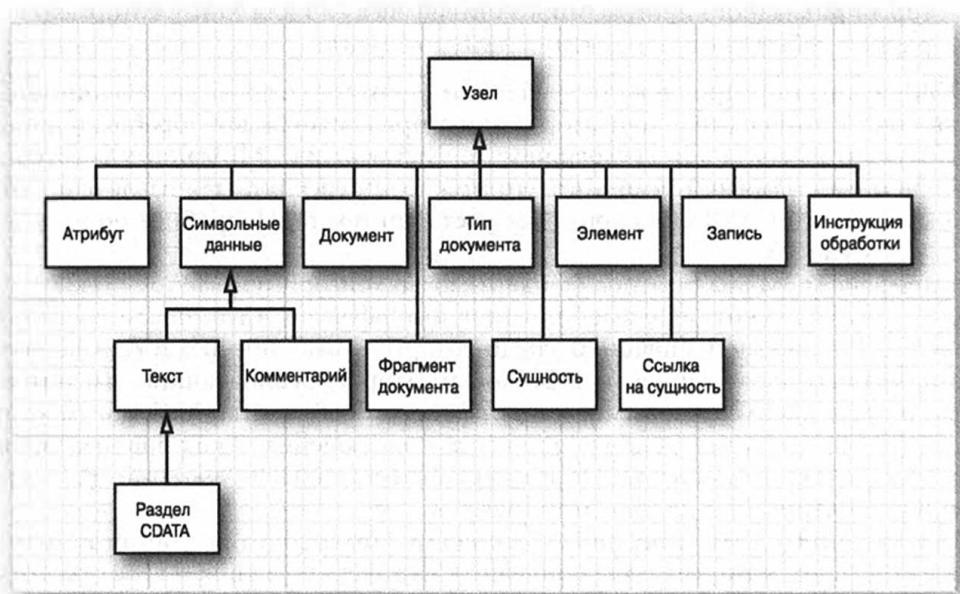


Рис. 3.1. Иерархия наследования интерфейса Node

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

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

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

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

Для извлечения элементов, дочерних по отношению к данному (ими могут быть подчиненные элементы, текст, комментарии или другие узлы), служит метод `getChildNodes()`, возвращающий набор данных типа `NodeList`. Этот тип

данных существовал еще до создания стандартной библиотеки коллекций в Java, и поэтому для него имеется другой протокол доступа. Метод `item()` возвращает элемент набора данных по указанному индексу, а метод `getLength()` — общее количество элементов. Таким образом, для перечисления всех дочерних элементов можно воспользоваться следующим кодом:

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

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

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

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

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

На рис. 3.2 схематически представлено дерево DOM — объектной модели упомянутого выше документа.

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

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

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

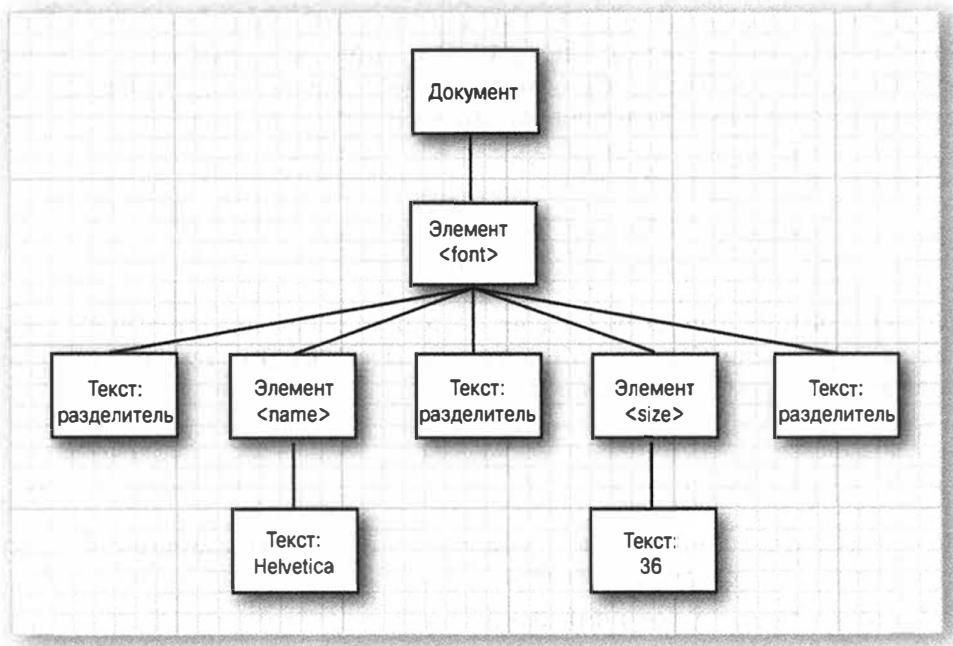


Рис. 3.2. Простое дерево DOM

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

```

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



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

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

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

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

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

Для перечисления атрибутов узла следует вызвать метод `getAttributes()`, возвращающий объект типа `NamedNodeMap`, который содержит объекты типа `Node`, описывающие атрибуты. Обход узлов в именованном отображении типа `NamedNodeMap` можно выполнить таким же образом, как и обход узлов в списке типа `NodeList`. В таком случае для извлечения имен атрибутов и их значений следует воспользоваться методами `getNodeName()` и `getNodeValue()`, как показано в приведенном ниже фрагменте кода.

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

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

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

Описанный выше способ анализа дерева DOM демонстрируется в примере программы, исходный код которой приведен в листинге 3.1. Для чтения XML-документа следует выбрать пункт меню `File`⇒`Open` (Файл⇒Открыть) и указать в диалоговом окне требуемый файл. После этого объект типа `DocumentBuilder` выполнит синтаксический анализ XML-данных и создаст объект типа `Document`. А далее этот объект будет представлен в рабочем окне программы в виде древовидной структуры (рис. 3.3).

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

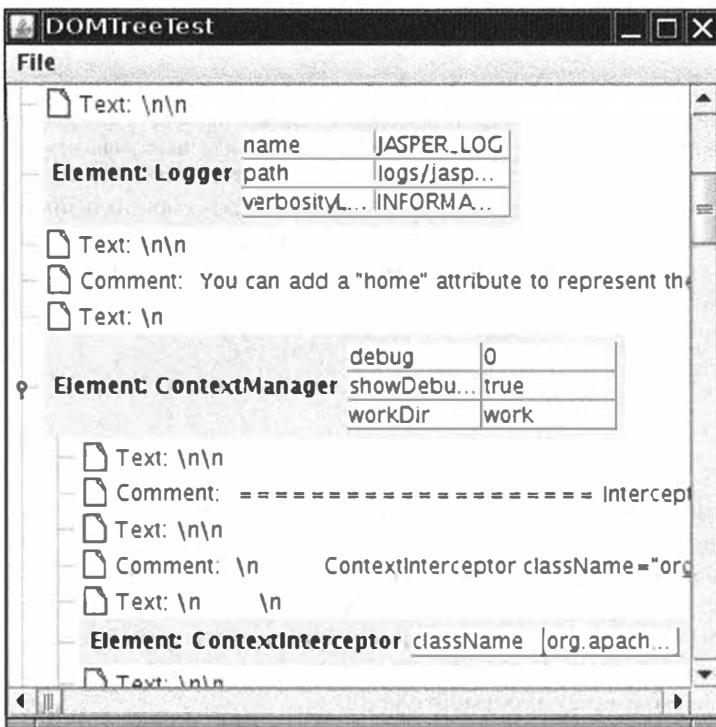


Рис. 3.3. Древовидная структура XML-документа, полученная в результате его синтаксического анализа

В данной программе применяются рассматриваемые в главе 10 способы отображения дерева и таблиц атрибутов. Класс DOMTreeModel реализует интерфейс TreeModel. Метод `getRoot()` возвращает корневой элемент документа, а метод `getChild()` извлекает список дочерних узлов и возвращает узел с заданным индексом. Средство воспроизведения ячеек дерева отображает следующее:

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

Листинг 3.1. Исходный код из файла dom/TreeViewer.java

```
1 package dom;
2
3 import java.awt.*;
4 import java.io.*;
5
6 import javax.swing.*;
7 import javax.swing.event.*;
```

```
8 import javax.swing.table.*;
9 import javax.swing.tree.*;
10 import javax.xml.parsers.*;
11
12 import org.w3c.dom.*;
13 import org.w3c.dom.CharacterData;
14
15 /**
16 * В этой программе XML-документ отображается в виде дерева
17 * @version 1.13 2016-04-27
18 * @author Cay Horstmann
19 */
20 public class TreeViewer
21 {
22     public static void main(String[] args)
23     {
24         EventQueue.invokeLater(() ->
25         {
26             JFrame frame = new DOMTreeFrame();
27             frame.setTitle("TreeViewer");
28             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29             frame.setVisible(true);
30         });
31     }
32 }
33
34 /**
35 * Этот фрейм содержит дерево, отображающее
36 * содержимое XML-документа
37 */
38 class DOMTreeFrame extends JFrame
39 {
40     private static final int DEFAULT_WIDTH = 400;
41     private static final int DEFAULT_HEIGHT = 400;
42
43     private DocumentBuilder builder;
44
45     public DOMTreeFrame()
46     {
47         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
48
49         JMenu fileMenu = new JMenu("File");
50         JMenuItem openItem = new JMenuItem("Open");
51         openItem.addActionListener(event -> openFile());
52         fileMenu.add(openItem);
53
54         JMenuItem exitItem = new JMenuItem("Exit");
55         exitItem.addActionListener(event -> System.exit(0));
56         fileMenu.add(exitItem);
57
58         JMenuBar menuBar = new JMenuBar();
59         menuBar.add(fileMenu);
60         setJMenuBar(menuBar);
61     }
62
63 /**
64 * Открыть файл и загрузить документ
65 */
```

```
66     public void openFile()
67     {
68         JFileChooser chooser = new JFileChooser();
69         chooser.setCurrentDirectory(new File("dom"));
70         chooser.setFileFilter(new javax.swing.filechooser.
71             .FileNameExtensionFilter("XML files", "xml"));
72         int r = chooser.showOpenDialog(this);
73         if (r != JFileChooser.APPROVE_OPTION) return;
74         final File file = chooser.getSelectedFile();
75
76         new SwingWorker<Document, Void>()
77         {
78             protected Document doInBackground() throws Exception
79             {
80                 if (builder == null)
81                 {
82                     DocumentBuilderFactory factory =
83                         DocumentBuilderFactory.newInstance();
84                     builder = factory.newDocumentBuilder();
85                 }
86                 return builder.parse(file);
87             }
88
89             protected void done()
90             {
91                 try
92                 {
93                     Document doc = get();
94                     JTree tree = new JTree(new DOMTreeModel(doc));
95                     tree.setCellRenderer(new DOMTreeCellRenderer());
96
97                     setContentPane(new JScrollPane(tree));
98                     validate();
99                 }
100                catch (Exception e)
101                {
102                    JOptionPane.showMessageDialog(DOMTreeFrame.this, e);
103                }
104            }
105        }.execute();
106    }
107 }
108
109 /**
110  * Эта модель дерева описывает древовидную
111  * структуру XML-документа
112 */
113 class DOMTreeModel implements TreeModel
114 {
115     private Document doc;
116
117     /**
118      * Строит модель дерева для документа
119      * @param doc Документ
120      */
121     public DOMTreeModel(Document doc)
122     {
123         this.doc = doc;
```

```
124 }
125
126 public Object getRoot()
127 {
128     return doc.getDocumentElement();
129 }
130
131 public int getChildCount(Object parent)
132 {
133     Node node = (Node) parent;
134     NodeList list = node.getChildNodes();
135     return list.getLength();
136 }
137
138 public Object getChild(Object parent, int index)
139 {
140     Node node = (Node) parent;
141     NodeList list = node.getChildNodes();
142     return list.item(index);
143 }
144
145 public int getIndexOfChild(Object parent, Object child)
146 {
147     Node node = (Node) parent;
148     NodeList list = node.getChildNodes();
149     for (int i = 0; i < list.getLength(); i++)
150         if (getChild(node, i) == child) return i;
151     return -1;
152 }
153
154 public boolean isLeaf(Object node)
155 {
156     return getChildCount(node) == 0;
157 }
158
159 public void valueForPathChanged(TreePath path,
160                                 Object newValue) {}
161 public void addTreeModelListener(TreeModelListener l) {}
162 public void removeTreeModelListener(TreeModelListener l) {}
163 }
164
165 /**
166 * Этот класс воспроизводит узел XML-документа
167 */
168 class DOMTreeCellRenderer extends DefaultTreeCellRenderer
169 {
170     public Component getTreeCellRendererComponent(JTree tree,
171                                               Object value, boolean selected,
172                                               boolean expanded, boolean leaf,
173                                               int row, boolean hasFocus)
174     {
175         Node node = (Node) value;
176         if (node instanceof Element)
177             return elementPanel((Element) node);
178
179         super.getTreeCellRendererComponent(tree, value, selected,
180                                           expanded, leaf, row, hasFocus);
181         if (node instanceof CharacterData)
```

```
182         setText(characterString((CharacterData) node));
183     else setText(node.getClass() + ": " + node.toString());
184     return this;
185 }
186
187 public static JPanel elementPanel(Element e)
188 {
189     JPanel panel = new JPanel();
190     panel.add(new JLabel("Element: " + e.getTagName()));
191     final NamedNodeMap map = e.getAttributes();
192     panel.add(new JTable(new AbstractTableModel()
193     {
194         public int getRowCount()
195         {
196             return map.getLength();
197         }
198
199         public int getColumnCount()
200         {
201             return 2;
202         }
203
204         public Object getValueAt(int r, int c)
205         {
206             return c == 0 ? map.item(r).getNodeName() :
207                         map.item(r).getNodeValue();
208         }
209     }));
210     return panel;
211 }
212
213 private static String characterString(CharacterData node)
214 {
215     StringBuilder builder = new StringBuilder(node.getData());
216     for (int i = 0; i < builder.length(); i++)
217     {
218         if (builder.charAt(i) == '\r')
219         {
220             builder.replace(i, i + 1, "\\\r");
221             i++;
222         }
223         else if (builder.charAt(i) == '\n')
224         {
225             builder.replace(i, i + 1, "\\\n");
226             i++;
227         }
228         else if (builder.charAt(i) == '\t')
229         {
230             builder.replace(i, i + 1, "\\\t");
231             i++;
232         }
233     }
234     if (node instanceof CDATASection)
235         builder.insert(0, "CDATASection: ");
236     else if (node instanceof Text)
237         builder.insert(0, "Text: ");
238     else if (node instanceof Comment)
239         builder.insert(0, "Comment: ");
```

```
240     return builder.toString();
241 }
242 }
243 }
```

javax.xml.parsers.DocumentBuilderFactory 1.4

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

javax.xml.parsers.DocumentBuilder 1.4

- **Document parse(File f)**
- **Document parse(String url)**
- **Document parse(InputStream in)**
Выполняют синтаксический анализ XML-документа, полученного из заданного файла, по указанному URL или из заданного потока ввода. Возвращают результат синтаксического анализа.

org.w3c.dom.Document 1.4

- **Element getDocumentElement()**
Возвращает корневой элемент разметки документа.

org.w3c.dom.Element 1.4

- **String getTagName()**
Возвращает имя элемента разметки.
- **String getAttribute(String name)**
Возвращает значение атрибута с заданным именем или пустую символьную строку, если такой атрибут отсутствует.

org.w3c.dom.Node 1.4

- **NodeList getChildNodes()**
Возвращает список, содержащий все дочерние узлы данного узла.
- **Node getFirstChild()**
- **Node getLastChild()**
Возвращают первый или последний дочерний узел данного узла. Если у данного узла отсутствуют дочерние узлы, возвращается пустое значение **null**.

org.w3c.dom.Node 1.4 (окончание)

- **Node getNextSibling()**
- **Node getPreviousSibling()**
Возвращает предыдущий родственный узел. Если у данного узла отсутствуют родственные узлы, возвращается пустое значение **null**.
- **Node getParentNode()**
Возвращает родительский узел данного узла или пустое значение **null**, если данный узел является узлом документа.
- **NamedNodeMap getAttributes()**
Возвращает отображение узлов, содержащее узлы типа **Attr** с описаниями всех атрибутов данного узла.
- **String getNodeName()**
Возвращает имя данного узла. Если узел относится к типу **Attr**, то возвращается имя атрибута.
- **String getNodeValue()**
Возвращает значение данного узла. Если узел относится к типу **Attr**, то возвращается значение атрибута.

org.w3c.dom.CharacterData 1.4

- **String getData()**
Возвращает текст, хранящийся в данном узле.

org.w3c.dom.NodeList 1.4

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

org.w3c.dom.NamedNodeMap 1.4

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

3.3. Проверка достоверности XML-документов

В предыдущем разделе описан способ обхода древовидной структуры DOM-документа. Но если следовать этому способу непосредственно, то потребуется приложить немало усилий для проверки ошибок программным путем. В этом случае придется не только организовать поиск и удаление лишних разделителей между элементами, но и проверить, содержит ли документ предполагаемые узлы. Рассмотрим в качестве примера следующий элемент разметки:

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

При чтении первого же дочернего узла неожиданно обнаруживается, что это текстовый узел, содержащий разделитель "\n". Пропуская текстовые узлы, нетрудно дойти до узла первого элемента, где нужно проверить, имеет ли его дескриптор имя name. Затем требуется выяснить, имеет ли он дочерний узел типа Text. После этого можно переместиться к следующему дочернему узлу без разделителя и выполнить такую же проверку. Но что делать, если составитель XML-документа изменит порядок расположения дочерних узлов или добавит еще один дочерний элемент? С одной стороны, для проверки всех возможных ошибок придется написать очень громоздкий код, а с другой — исключить такую проверку было бы слишком опрометчиво.

Правда, к числу главных преимуществ XML-анализатора относится его способность автоматически проверять корректность структуры документа. В таком случае анализ XML-документа значительно упрощается. Так, если известно, что элемент разметки font успешно прошел проверку, то несложно получить два дочерних узла, привести их к типу Text, а затем извлечь текстовые данные без дополнительной проверки.

Для указания структуры документа можно предоставить определение DTD или XML Schema. Определение DTD или XML Schema содержит правила, регламентирующие структуру документа. Оно задает допустимые дочерние узлы элементов и атрибутов каждого элемента. Например, определение DTD может содержать следующее правило:

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

Это правило выражает следующее ограничение: у элемента разметки font всегда должны быть два дочерних узла: name и size. А средствами языка XML Schema то же самое ограничение записывается следующим образом:

```
<xsd:element name="font">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="xsd:int"/>
  </xsd:sequence>
</xsd:element>
```

Язык XML Schema позволяет формулировать более изощренные условия проверки достоверности, чем определения DTD. Например, элемент разметки size должен содержать целочисленное значение. В отличие от определения DTD,

в XML Schema используется синтаксис XML, что упрощает обработку файлов со схемами XML-документов.

В следующем разделе подробно обсуждаются определения DTD, а затем вкратце рассматриваются основные средства поддержки XML Schema. После этого будет представлен пример, наглядно демонстрирующий, насколько проверка достоверности XML-документов упрощает их обработку.

3.3.1. Определения типов документов

Существует несколько способов предоставить определение типа документа (DTD). В частности, определение DTD можно ввести в начале XML-документа, как показано ниже.

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

Как видите, эти правила заключаются в квадратные скобки объявления DOCTYPE. Тип документа должен соответствовать имени корневого элемента (в данном примере — configuration). Размещать определения DTD в самом документе неудобно, поскольку они могут быть очень длинными. Следовательно, определения DTD имеет смысл хранить в отдельном файле. А для связывания определений DTD с XML-документами можно использовать объявления SYSTEM, в которых указываются имена внешних файлов,

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



Внимание! Если для указания внешнего файла с определением DTD служит относительный URL (например, "config.dtd"), то вместо потока ввода типа `InputStream` синтаксическому анализатору следует предоставить объект типа `File` или `URL`. Если же требуется синтаксический анализ данных из потока ввода, то синтаксическому анализатору следует предоставить определитель сущностей, как поясняется в следующем далее примечании.

И наконец, можно воспользоваться механизмом обозначения хорошо известных определений DTD, унаследованным из языка SGML, как в приведенном ниже примере. Если XML-процессору известен способ обнаружения DTD с помощью идентификатора PUBLIC, то обращаться по указанному URL совсем не обязательно.

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

 **На заметку!** Если используется DOM-анализатор и требуется организовать поддержку идентификатора PUBLIC, следует вызвать метод `setEntityResolver()` из класса `DocumentBuilder`. С помощью этого метода устанавливается экземпляр класса, реализующего интерфейс `EntityResolver`. В данном интерфейсе определен единственный метод `resolveEntity()`. Ниже приведена типичная реализация этого метода.

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

Источник входных данных можно создать из потока ввода типа `InputStream, Reader` или символьной строки.

Рассмотрим теперь различные правила, которые могут задаваться в определении DTD. Правило ELEMENT задает дочерние узлы данного элемента в виде регулярного выражения, составляющие которого перечислены в табл. 3.1.

Таблица 3.1. Правила для содержимого документа

Правило	Назначение
E^*	0 или больше вхождений элемента E
E^+	1 или больше вхождений элемента E
$E?$	0 или 1 вхождение элемента E
$E_1 E_2 \dots E_n$	Один из элементов E_1, E_2, \dots, E_n
E_1, E_2, \dots, E_n	Элемент E_1 , после которого следуют элементы E_2, \dots, E_n
#PCDATA	Текст
(#PCDATA $E_1 E_2 \dots E_n$) *	0 или больше вхождений текста и последовательность элементов E_1, E_2, \dots, E_n в любом порядке (смешанное содержимое)
ANY	Любой дочерний узел
EMPTY	Дочерние узлы отсутствуют

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

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

Согласно приведенным ниже правилам, шрифт описывается именем, после которого следует размер шрифта. Имя и размер шрифта являются текстовыми элементами.

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

Сокращение PCDATA обозначает проанализированные символьные данные. Данные называются *проанализированными*, поскольку синтаксический анализатор

обрабатывает текстовую строку и ищет знак <, обозначающий начало нового дескриптора, или же знак &, обозначающий начало сущности. Спецификация элемента может содержать регулярные выражения, в том числе вложенные и сложные. В качестве примера ниже приведено правило, описывающее структуру главы в книге. Каждая глава (*chapter*) начинается с введения (*intro*), после которого следует один или несколько разделов, состоящих из заголовка (*heading*), одного или нескольких абзацев (*para*), рисунков (*image*), таблиц (*table*) или примечаний (*note*).

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

Но далеко не всегда правила обеспечивают достаточную гибкость. Очень часто элементы содержат текст, и тогда допускаются только два варианта. Во-первых, в составе элемента допускается наличие только текста:

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

И во-вторых, элемент может содержать любое сочетание *текста и дескрипторов*, располагаемых в произвольном порядке:

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

А другие правила типа #PCDATA не допускаются. Например, приведенное ниже выражение неверно. Такие правила следует переписать заново, введя еще один элемент разметки *caption*, содержащий надпись, или допустив любое сочетание элементов разметки *image* и текста.

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

Подобное ограничение упрощает работу XML-анализатора при синтаксическом анализе *смешанного содержимого* (текста и дескрипторов). Но в этом случае содержимое становится неконтролируемым, поэтому рекомендуется составлять такие определения DTD, которые содержат только элементы разметки или же ничего, кроме текста.



На заметку! Не совсем верно считать, что в правилах DTD можно указывать произвольные регулярные выражения. XML-анализатор может отклонить сложные наборы правил, которые до определенного момента не дают однозначного результата. Примером тому служит регулярное выражение $(x, y) | (x, z)$. Обнаружив в нем элемент *x*, анализатор не сможет сразу выяснить, какой из двух альтернативных вариантов выбрать. Это выражение следует переписать в следующем виде: $(x, (y|z))$. Но некоторые определения нельзя изменить, например $(x,y)*x?$. Синтаксический анализатор из библиотеки Java XML не выдает никаких предупреждений при обнаружении подобных определений DTD. Он просто выбирает первый совпадающий вариант, что может привести к неверной интерпретации правильных входных данных. На практике определения, подобные приведенным выше, встречаются крайне редко, поскольку большинство определений DTD очень просты.

Для описания допустимых атрибутов элементов разметки используется приведенный ниже синтаксис. В табл. 3.2 приведены допустимые типы атрибутов, а в табл. 3.3 — синтаксис поведения атрибутов по умолчанию.

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

Таблица 3.2. Типы атрибутов

Тип	Назначение
CDATA	Произвольная символьная строка
(A₁ A₂ ... A_n)	Один из строковых атрибутов A ₁ , A ₂ , ..., A _n
NMTOKEN, NMTOKENS	Одна или несколько лексем, соответствующих имени
ID	Однозначный идентификатор
IDREF, IDREFS	Одна или несколько ссылок на однозначный идентификатор
ENTITY, ENTITIES	Одна или несколько непроанализированных сущностей

Таблица 3.3. Поведение атрибутов по умолчанию

Поведение по умолчанию	Назначение
#REQUIRED	Атрибут является обязательным
#IMPLIED	Атрибут не является обязательным
A	Атрибут не является обязательным; анализатор возвращает значение A , если атрибут не указан
#FIXED A	Атрибут не должен быть указан или должен быть равен A ; но в любом случае анализатор возвращает значение A

Ниже представлены два типичных примера спецификации атрибутов.

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

В первом примере для элемента разметки `font` указан атрибут `style`, который может иметь четыре допустимых значения. По умолчанию используется значение `plain`. А во втором примере для элемента разметки `size` указан атрибут `unit`, который может содержать любую последовательность символов.



На заметку! Для описания данных рекомендуется применять элементы разметки, а не атрибуты. В соответствии с этой рекомендацией стиль шрифта должен содержаться в отдельном элементе разметки, например, в следующем виде: `<style>plain</style>...`. Но атрибуты обладают несомненным преимуществом при использовании перечислений, потому что анализатор может проверять допустимость тех или иных переменных. Так, если стиль шрифта является атрибутом, то анализатор проверяет наличие найденного стиля среди четырех указанных допустимых значений и выбирает значение по умолчанию, если ничего не указано.

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

Тип атрибута `NMTOKEN` (т.е. лексема имени) аналогичен типу `CDATA`, но в нем не допускается использование большинства символов, отличающихся от букв и цифр, а также разделителей в виде внутренних пробелов. Синтаксический анализатор удаляет все начальные и конечные пробелы как разделители. Тип атрибута `NMTOKENS` представляет собой список лексем имен, разделяемых пробелами.

Тип атрибута `ID` означает лексему имени, однозначную для данного документа. Однозначность лексемы имени проверяется синтаксическим анализатором.

Применение данного типа атрибута демонстрируется в рассматриваемом далее примере программы. Тип атрибута IDREF означает ссылку на идентификатор, уже существующий в данном документе, наличие которого также проверяется синтаксическим анализатором. А тип атрибута IDREFS обозначает список ссылок на идентификаторы.

Атрибут типа ENTITY указывает на “непроанализированную внешнюю сущность”. Этот тип унаследован от SGML и редко применяется на практике. В спецификации языка XML, доступной по адресу www.xml.com/axml/axml.html, приводится пример применения подобного типа атрибута.

Определение DTD может также содержать определения *сущностей*, или сокращений, которые заменяются в процессе синтаксического анализа. Характерный пример применения сущностей можно найти в описании пользовательского интерфейса веб-браузера Firefox. Данное описание отформатировано в соответствии с требованиями XML и содержит определения сущностей, подобные приведенному ниже.

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

Далее в тексте документа могут встретиться ссылки на эти сущности, как показано в следующем примере:

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

В таком случае синтаксический анализатор заменяет ссылку на сущность замещающей строкой. Для internaцionalизации прикладной программы потребуется лишь изменить определение самой сущности. Другие примеры применения сущностей более сложны и менее распространены. Дополнительные сведения по данному вопросу можно найти в спецификации языка XML по указанному выше адресу.

На этом краткое введение в определения DTD завершается. Рассмотрим далее способы настройки синтаксического анализатора, чтобы воспользоваться всеми преимуществами DTD. Для этого нужно сначала сообщить фабрике построителей документов о необходимости активизировать проверку следующим образом:

```
factory.setValidating(true);
```

Все построители документов, производимые этой фабрикой, проверяют соответствие входных данных определению DTD. Наиболее полезным результатом такой проверки является игнорирование всех разделителей в элементе разметки. Рассмотрим в качестве примера следующий фрагмент XML-разметки:

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

Синтаксический анализатор, не выполняяший проверку, возвращает все разделители между элементами разметки font, name и size. Он поступает так потому, что ему неизвестно, какое из следующих правил описывает дочерние узлы элемента разметки font:

```
(name,size)
(#PCDATA,name,size)*
```

или

ANY

Если же в определении DTD указано правило (**name**, **size**), то синтаксическому анализатору должно быть известно, что разделитель этих элементов не относится к тексту. Построитель документа не будет учитывать разделители в узлах текста, если вызвать следующий метод:

```
factory.setIgnoringElementContentWhitespace(true);
```

Теперь нет никаких сомнений, что узел **font** имеет два дочерних узла. Следовательно, нет никаких оснований организовывать приведенный ниже громоздкий цикл.

```
for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    if (child instanceof Element)
    {
        Element childElement = (Element) child;
        if (childElement.getTagName().equals("name")) . . .
        else if (childElement.getTagName().equals("size")) . . .
    }
}
```

Вместо этого для доступа к первому и второму дочерним узлам можно написать следующий фрагмент кода:

```
Element nameElement = (Element) children.item(0);
Element sizeElement = (Element) children.item(1);
```

Именно в этих случаях особенно полезны определения DTD. Как видите, при использовании определений DTD в программу уже не нужно включать сложные фрагменты кода, выполняющие проверку, потому что синтаксический анализатор проделает эту работу при получении документа.



Совет! Приступая к обработке XML-документов, многие программисты игнорируют описания DTD и пытаются самостоятельно анализировать древовидную структуру модели DOM. Чтобы убедиться в неэффективности такого подхода, достаточно только взглянуть на два приведенных выше варианта написания кода для обработки XML-документа.

Если синтаксический анализатор сообщает об ошибке, прикладная программа должна каким-то образом отреагировать на это, в частности, зарегистрировать ошибку, сообщить о ней пользователю или сгенерировать исключение, чтобы прекратить синтаксический анализ. Поэтому при использовании средств проверки содержимого XML-документа следует также установить обработчик исключений, создав объект, класс которого реализует интерфейс **ErrorHandler**. В этом интерфейсе объявлены следующие методы:

```
void warning(SAXParseException exception)
void error(SAXParseException exception)
void fatalError(SAXParseException exception)
```

А для установки обработчика исключений служит следующий метод **setErrorHandler()** из класса **DocumentBuilder**:

```
builder.setErrorHandler(handler);
```

javax.xml.parsers.DocumentBuilder 1.4

- **void setEntityResolver(EntityResolver resolver)**

Устанавливает определитель сущностей, упоминаемых в анализируемых XML-документах.

- **void setErrorHandler(ErrorHandler handler)**

Устанавливает обработчик исключений для выдачи предупреждений и сообщений об ошибках, возникающих при синтаксическом анализе XML-документов.

org.xml.sax.EntityResolver 1.4

- **public InputSource resolveEntity(String publicID, String systemID)**

Возвращает источник вводимых данных, содержащий данные, определяемые заданными идентификаторами, или пустое значение **null**, указывающее на то, что определителю сущностей неизвестно, как обработать данное конкретное имя. Параметр **publicID** может принимать пустое значение **null**, если открытые идентификаторы не предоставляются.

org.xml.sax.InputSource 1.4

- **InputSource(InputStream in)**

- **InputSource(Reader in)**

- **InputSource(String systemID)**

Создают источник вводимых данных на основании указанного потока ввода, потока чтения или системного идентификатора (обычно это относительный или абсолютный URL).

org.xml.sax.ErrorHandler 1.4

- **void fatalError(SAXParseException exception)**

- **void error(SAXParseException exception)**

- **void warning(SAXParseException exception)**

Эти методы следует переопределить для создания собственного обработчика неустранимых ошибок, устранимых ошибок или предупреждений.

org.xml.sax.SAXParseException 1.4

- **int getLineNumber()**

- **int getColumnNumber()**

Возвращают номер строки или столбца в конце вводимых данных, при обработке которых возникло исключение.

javax.xml.parsers.DocumentBuilderFactory 1.4

- **boolean isvalidating()**
- **void setValidating(boolean value)**

Возвращают или устанавливают свойство **validating** для фабрики. Если это свойство принимает логическое значение **true**, то созданные фабрикой синтаксические анализаторы будут выполнять проверку входных данных.

- **boolean isIgnoringElementContentWhitespace()**
- **void setIgnoringElementContentWhitespace(boolean value)**

Получают или устанавливают значение, определяющее, следует ли игнорировать разделители между элементами разметки. Логическое значение **true** указывает на то, что созданные фабрикой синтаксические анализаторы будут игнорировать разделители в том случае, если для элемента разметки не задано смешанное содержимое (т.е. сочетание элементов разметки с атрибутами типа #PCDATA).

3.3.2. Схема XML-документов

Синтаксис языка XML Schema сложнее, чем у определений DTD, поэтому рассмотрим лишь самые основные его элементы. Дополнительные сведения о нем можно найти в учебном руководстве, доступном по адресу <http://www.w3.org/TR/xmlschema-0>. Чтобы включить в документ ссылку на файл схемы типа XML Schema, в корневом элементе следует указать соответствующие атрибуты, как показано в приведенном ниже примере.

```
<?xml version="1.0"?>
<configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="config.xsd">
    .
    .
</configuration>
```

В данном примере объявления указывается, что при проверке документа должен использоваться файл схемы config.xsd. Если же в XML-документе применяются пространства имен, то синтаксис немного усложняется. Подробнее об этом можно узнать из учебного руководства по указанному выше адресу. (Префикс **xsi** обозначает *псевдоним пространства имен*, как поясняется далее, в разделе 3.5.)

Схема определяет тип каждого элемента. Это может быть *простой тип* (символьная строка с ограничениями на форматирование) или же *сложный тип*. Некоторые простые типы встроены в XML Schema. Примеры таких типов приведены ниже.

```
xsd:string
xsd:int
xsd:boolean
```



На заметку! Здесь и далее используется префикс **xsd:**, обозначающий пространство имен XML Schema Definition. Некоторые авторы применяют для этой же цели префикс **xs:**.

По желанию можно определить собственные простые типы. Ниже приведен пример определения перечислимого типа.

```
<xsd:simpleType name="StyleType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="PLAIN" />
    <xsd:enumeration value="BOLD" />
    <xsd:enumeration value="ITALIC" />
    <xsd:enumeration value="BOLD_ITALIC" />
  </xsd:restriction>
</xsd:simpleType>
```

Определяя элемент разметки, следует указать его тип следующим образом:

```
<xsd:element name="name" type="xsd:string"/>
<xsd:element name="size" type="xsd:int"/>
<xsd:element name="style" type="StyleType"/>
```

Тип ограничивает возможные варианты содержимого элемента. Например, проверка следующих элементов разметки даст положительный результат:

```
<size>10</size>
<style>PLAIN</style>
```

А приведенные ниже элементы разметки будут отвергнуты синтаксическим анализатором.

```
<size>default</size>
<style>SLANTED</style>
```

Простые типы можно объединять в сложные:

```
<xsd:complexType name="FontType">
  <xsd:sequence>
    <xsd:element ref="name"/>
    <xsd:element ref="size"/>
    <xsd:element ref="style"/>
  </xsd:sequence>
</xsd:complexType>
```

где *FontType* — последовательность элементов разметки *name*, *size* и *style*. В данном определении использован атрибут *ref*, для которого ссылки на определения находятся в схеме. Допускаются также вложенные определения, как в приведенном ниже примере.

```
<xsd:complexType name="FontType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="xsd:int"/>
    <xsd:element name="style" type="StyleType">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="PLAIN" />
          <xsd:enumeration value="BOLD" />
          <xsd:enumeration value="ITALIC" />
          <xsd:enumeration value="BOLD_ITALIC" />
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

Обратите внимание на то, что для элемента `style` использовано *анонимное определение типа*. Языковая конструкция `xsd:sequence` является аналогом операции сцепления определений DTD, а конструкция `xsd:choice` равнозначна логической операции `|`. Так, приведенная ниже схема разметки заменяет тип `email | phone` в определении DTD.

```
<xsd:complexType name="contactinfo">
  <xsd:choice>
    <xsd:element ref="email"/>
    <xsd:element ref="phone"/>
  </xsd:choice>
</xsd:complexType>
```

Для повторяющихся элементов можно использовать атрибуты `minoccurs` и `maxOccurs`. Например, аналогом типа `item*` в определении DTD является следующая схема разметки:

```
<xsd:element name="item" type=". . ." minoccurs="0"
  maxOccurs="unbounded">
```

Для определения атрибутов в разметку определений `complexType` следует ввести элементы `xsd:attribute`:

```
<xsd:element name="size">
  <xsd:complexType>
    . . .
    <xsd:attribute name="unit" type="xsd:string"
      use="optional" default="см"/>
  </xsd:complexType>
</xsd:element>
```

Приведенной выше схеме разметки равнозначен следующий оператор в определении DTD:

```
<!ATTLIST size unit CDATA #IMPLIED "см">
```

Определения элемента и типа схемы разметки размещаются в элементе `xsd:schema` следующим образом:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  . . .
</xsd:schema>
```

Синтаксический анализ XML-документа, разметка которого определяется по заданной схеме, выполняется практически так же, как и синтаксический анализ XML-документа, для разметки которого служит определение DTD, за исключением нескольких перечисленных ниже отличий.

1. Активизировать средства поддержки пространств имен в XML-файлах, даже если они не используются в XML-документах.

```
factory.setNamespaceAware(true);
```

2. Подготовить фабрику для обработки схем разметки с помощью следующих выражений:

```
final String JAXP_SCHEMA_LANGUAGE =
  "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
final String W3C_XML_SCHEMA = "http://www.w3.org/2001/XMLSchema";
factory.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
```

3. Анализатор не отвергает разделители в элементах разметки. Это серьезное упущение, относительно которого существует разногласие, является ли оно ошибкой. В рассматриваемом далее примере схемы разметки компоновки из листинга 3.4 демонстрируется один из способов разрешения подобного разногласия.

3.3.3. Практический пример применения XML-документов

В этом разделе рассматривается пример применения XML-документа на практике. Как отмечалось в главе 12 первого тома настоящего издания, диспетчер сеточно-контейнерной компоновки типа `GridBagLayout` лучше всего подходит для размещения компонентов библиотеки `Swing`. Но многие не решаются применять его не столько в силу его сложности, сколько из-за большого объема рутинной работы. При использовании данного диспетчера компоновки вместо создания большого количества повторяющихся фрагментов кода гораздо удобнее было бы разместить все инструкции в текстовом файле. В этом разделе показано, как пользоваться XML-разметкой для описания сеточно-контейнерной компоновки и синтаксического анализа файлов с подобным описанием.

Сеточно-контейнерная компоновка состоит из рядов и столбцов, подобно HTML-таблице. Поэтому ее можно описать в виде последовательности рядов, составленных из отдельных ячеек:

```
<gridbag>
  <row>
    <cell>...</cell>
    <cell>...</cell>
    .
  </row>
  <row>
    <cell>...</cell>
    <cell>...</cell>
    .
  </row>
  .
</gridbag>
```

В файле `gridbag.dtd` указываются следующие правила:

```
<!ELEMENT gridbag (row)*>
<!ELEMENT row (cell)*>
```

Некоторые ячейки могут распространяться на несколько рядов и столбцов. В диспетчере сеточно-контейнерной компоновки это достигается путем установки ограничений в виде значений больше 1 в полях `gridwidth` и `gridheight`. А в XML-разметке используются атрибуты с теми же именами следующим образом:

```
<cell gridwidth="2" gridheight="2">
```

Аналогично вводятся атрибуты, определяющие другие ограничения: `fill`, `anchor`, `.gridx`, `.gridy`, `weightx`, `weighty`, `ipadx` и `ipady`, как показано ниже. (Ограничение `insets` не используется в данном примере, поскольку его нельзя

представить с помощью простого типа, хотя организовать его поддержку не составит особого труда.)

```
<cell fill="HORIZONTAL" anchor="NORTH">
```

Для большинства атрибутов используются те же значения по умолчанию, что и в конструкторе без аргументов класса `GridBagConstraints`:

```
<!ATTLIST cell gridwidth CDATA "1">
<!ATTLIST cell gridheight CDATA "1">
<!ATTLIST cell fill (NONE|BOTH|HORIZONTAL|VERTICAL) "NONE">
<!ATTLIST cell anchor (CENTER|NORTH|NORTHEAST|EAST
    |SOUTHEAST|SOUTH|SOUTHWEST|WEST|NORTHWEST) "CENTER">
...

```

Значения `gridx` и `gridy` задаются и обрабатываются иначе, поскольку указывать их вручную оказывается неудобно, да и чревато ошибками. По этим причинам они указываются как необязательные следующим образом:

```
<!ATTLIST cell gridx CDATA #IMPLIED>
<!ATTLIST cell gridy CDATA #IMPLIED>
```

Если же эти значения не указаны, то в программе они будут определены следующим образом: в нулевом столбце по умолчанию будет задано нулевое значение `gridx`, а во всех остальных столбцах — предыдущее значение `gridx` плюс предыдущее значение `gridwidth`. По умолчанию значение `gridy` всегда совпадает с номером ряда. Поэтому значения `gridx` и `gridy`, как правило, указывать не нужно. Но если компонент простирается на несколько столбцов, то значение `gridx` придется указывать, если требуется пропустить данный компонент.



На заметку! Знатоки сеточно-контейнерной компоновки скажут, что для автоматического определения значений `gridx` и `gridy` можно было бы воспользоваться константами `RELATIVE` и `REMAINDER`. Мы попытались это сделать, но получить такое же расположение элементов, как и в диалоговом окне выбора шрифта, приведенном на рис. 3.4, нам не удалось. После внимательного изучения исходного кода класса `GridBagLayout` мы пришли к выводу, что этот класс просто не в состоянии восстанавливать абсолютные позиции.

Во время работы рассматриваемой здесь программы анализируются атрибуты и задаются ограничения для сеточно-контейнерной компоновки. Например, для определения ширины сетки в программе служит следующая строка кода:

```
constraints.gridwidth =
    Integer.parseInt(e.getAttribute("gridwidth"));
```

Программе не нужно беспокоиться о пропущенных атрибутах, поскольку синтаксический анализатор автоматически предоставит значение по умолчанию, если оно не указано в документе.

Чтобы выяснить, были ли указаны атрибуты `gridx` и `gridy`, вызывается метод `getAttribute()` и проверяется, возвращает ли он пустую символьную строку:

```
String value = e.getAttribute("gridy");
if (value.length() == 0) // использовать по умолчанию
    constraints.gridy = r;
else
    constraints.gridy = Integer.parseInt(value);
```

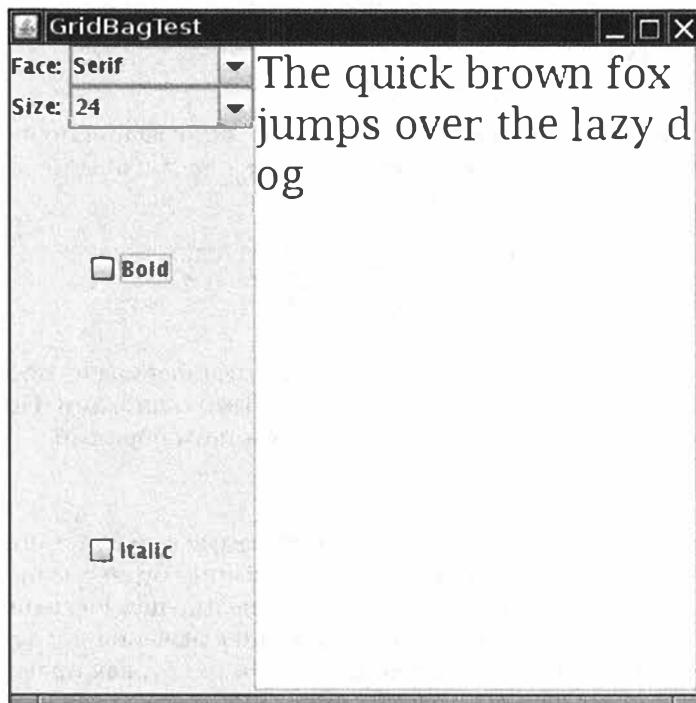


Рис. 3.4. Диалоговое окно выбора шрифта, расположение компонентов в котором определяется в XML-документе

Оказывается, что удобнее предоставить пользователю возможность указывать произвольные объекты в ячейках. Это позволяет задавать даже такие типы, как, например, границы, которые не относятся к компонентам. Нужно лишь, чтобы объекты принадлежали классу, отвечающему следующим требованиям к компонентам JavaBeans: он должен иметь конструктор без аргументов и свойства, задаваемые парами методов получения и установки. Компонент JavaBeans определяется именем класса и свойствами следующим образом:

```
<!ELEMENT bean (class, property*)>
<!ELEMENT class (#PCDATA)>
```

Свойство содержит имя и значение, как показано ниже.

```
<!ELEMENT property (name, value)>
<!ELEMENT name (#PCDATA)>
```

Значение может быть целым, логическим, строковым или другим компонентом JavaBeans:

```
<!ELEMENT value (int|string|boolean|bean)>
<!ELEMENT int (#PCDATA)>
<!ELEMENT string (#PCDATA)>
<!ELEMENT boolean (#PCDATA)>
```

Ниже в качестве характерного примера приведена разметка компонента JLabel со свойством text, которое принимает строковое значение "Face: ".

```
<bean>
<class>javax.swing.JLabel</class>
<property>
  <name>text</name>
  <value><string>Face: </string></value>
</property>
</bean>
```

На первый взгляд, излишне заключать символьную строку в дескрипторы <string>. Почему бы не указывать правило #PCDATA для символьных строк, а дескрипторы применять только для других типов? Потому что в таком случае придется использовать смешанное содержание и менее строгое правило для элемента value:

```
<!ELEMENT value (#PCDATA|int|boolean|bean)*>
```

Значение свойства задается в рассматриваемой здесь программе с помощью класса BeanInfo, где перечисляются описатели свойств компонента JavaBeans. Сначала осуществляется поиск свойства по заданному имени, а затем вызывается метод его установки с требующимся значением в качестве параметра.

Прочитав описание пользовательского интерфейса, программа получает достаточно сведений для создания компонентов пользовательского интерфейса и их расположения. Но такой интерфейс, безусловно, не будет действовать, поскольку в нем отсутствуют обработчики событий. Для ввода обработчиков событий придется разместить соответствующие компоненты. Поэтому для каждого компонента JavaBeans поддерживается необязательный атрибут ID:

```
<!ATTLIST bean id ID #IMPLIED>
```

В качестве примера ниже приведена разметка комбинированного списка с указанием его идентификатора.

```
<bean id="face">
<class>javax.swing.JComboBox</class>
</bean>
```

Напомним, что синтаксический анализатор проверяет однозначность идентификаторов. А присоединить обработчики событий к компонентам пользовательского интерфейса можно следующим образом:

```
gridbag = new GridBagPanel("fontdialog.xml");
setContentPane(gridbag);
JComboBox face = (JComboBox) gridbag.get("face");
face.addListener(listener);
```



На заметку! В рассматриваемом здесь примере XML-разметка служит только для описания компоновки компонента, а вводить обработчики событий в код Java можно самостоятельно. Но можно пойти еще дальше, включив код в описание, составленное в формате XML. Для этой цели лучше всего подходит такой язык написания сценариев, как JavaScript, а также его интерпретатор Nashorn, описываемый в главе 8.

В примере программы, исходный код которой приведен в листинге 3.2, демонстрируется применение класса `GridBagPane` для выполнения всей трудоемкой и рутинной работы по подготовке сеточно-контейнерной компоновки. Расположение компонентов определяется в XML-файле, содержимое которого приведено в листинге 3.4, а внешний вид данной компоновки показан на рис. 3.4. В данной программе инициализируются только комбинированные списки, которые слишком сложны, чтобы применять к ним механизм установки свойств компонентов JavaBeans, поддерживаемый в классе `GridBagPane`, а также присоединяются обработчики событий. В классе `GridBagPane`, исходный код которого приведен в листинге 3.3, выполняется синтаксический анализ разметки из XML-файла, создаются и располагаются компоненты в рабочем окне программы. А в листинге 3.5 представлено определение DTD.

В рассматриваемой здесь программе поддерживается также обработка схемы XML-документа вместо определения DTD. Для этого данную программу нужно запустить с указанием файла схемы, например, так, как показано ниже. Сама же схема приведена в листинге 3.6.

```
java GridBagTest fontdialog-schema.xml
```

В данной программе демонстрируется типичный пример применения формата XML, который позволяет выражать достаточно сложные взаимосвязи между объектами. А XML-анализатор берет на себя всю рутинную работу по проверке достоверности документа и установке значений по умолчанию.

Листинг 3.2. Исходный код из файла `read/GridBagTest.java`

```

1 package read;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.io.*;
6 import javax.swing.*;
7
8 /**
9  * В этой программе демонстрируется применение XML-документа
10 * для описания сеточно-контейнерной компоновки
11 * @version 1.12 2016-04-27
12 * @author Cay Horstmann
13 */
14 public class GridBagTest
15 {
16     public static void main(String[] args)
17     {
18         EventQueue.invokeLater(() ->
19         {
20             JFileChooser chooser = new JFileChooser(".");
21             chooser.showOpenDialog(null);
22             File file = chooser.getSelectedFile();
23             JFrame frame = new FontFrame(file);
24             frame.setTitle("GridBagTest");
25             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26             frame.setVisible(true);
27         });
28     }
29 }
```

```
30
31 /**
32  * Этот фрейм содержит диалоговое окно для выбора шрифтов,
33  * описываемое в XML-файле
34  * @param filename Файл, содержащий компоненты пользовательского,
35  *                  интерфейса, располагаемые в диалоговом окне
36 */
37 class FontFrame extends JFrame
38 {
39     private GridBagPane gridbag;
40     private JComboBox<String> face;
41     private JComboBox<String> size;
42     private JCheckBox bold;
43     private JCheckBox italic;
44
45     @SuppressWarnings("unchecked")
46     public FontFrame(File file)
47     {
48         gridbag = new GridBagPane(file);
49         add(gridbag);
50
51         face = (JComboBox<String>) gridbag.get("face");
52         size = (JComboBox<String>) gridbag.get("size");
53         bold = (JCheckBox) gridbag.get("bold");
54         italic = (JCheckBox) gridbag.get("italic");
55
56         face.setModel(new DefaultComboBoxModel<String>(new String[]
57             { "Serif", 54 "SansSerif", "Monospaced",
58               "Dialog", "DialogInput" }));
59
60         size.setModel(new DefaultComboBoxModel<String>(new String[]
61             { "8", 57 "10", "12", "15", "18", "24", "36", "48" }));
62
63         ActionListener listener = event -> setSample();
64
65         face.addActionListener(listener);
66         size.addActionListener(listener);
67         bold.addActionListener(listener);
68         italic.addActionListener(listener);
69
70         setSample();
71         pack();
72     }
73
74 /**
75  * This Этот метод выделяет образец текста выбранным шрифтом
76  */
77 public void setSample()
78 {
79     String fontFace = face.getItemAt(face.getSelectedIndex());
80     int fontSize = Integer.parseInt(size.getItemAt(
81                                     size.getSelectedIndex()));
82     JTextArea sample = (JTextArea) gridbag.get("sample");
83     int fontStyle = (bold.isSelected() ? Font.BOLD : 0)
84         + (italic.isSelected() ? Font.ITALIC : 0);
85
86     sample.setFont(new Font(fontFace, fontStyle, fontSize));
87     sample.repaint();
88 }
89 }
```

Листинг 3.3. Исходный код из файла `read/GridBagPane.java`

```
1 package read;
2
3 import java.awt.*;
4 import java.beans.*;
5 import java.io.*;
6 import java.lang.reflect.*;
7 import javax.swing.*;
8 import javax.xml.parsers.*;
9 import org.w3c.dom.*;
10
11 /**
12 * Для описания сеточно-контейнерной компоновки отдельных
13 * компонентов этой панели служит XML-файл
14 */
15 public class GridBagPane extends JPanel
16 {
17     private GridBagConstraints constraints;
18
19     /**
20      * Строит панель по сеточно-контейнерной компоновке
21      * @param filename Имя XML-файла, описывающего компоненты
22      *                  панели и их расположение
23     */
24     public GridBagPane(File file)
25     {
26         setLayout(new GridBagLayout());
27         constraints = new GridBagConstraints();
28
29         try
30         {
31             DocumentBuilderFactory factory =
32                 DocumentBuilderFactory.newInstance();
33             factory.setValidating(true);
34
35             if (file.toString().contains("-schema"))
36             {
37                 factory.setNamespaceAware(true);
38                 final String JAXP_SCHEMA_LANGUAGE = "http://
39                     java.sun.com/xml/jaxp/properties/schemaLanguage";
40                 final String W3C_XML_SCHEMA =
41                     "http://www.w3.org/2001/XMLSchema";
42                 factory.setAttribute(JAXP_SCHEMA_LANGUAGE,
43                                     W3C_XML_SCHEMA);
44             }
45
46             factory.setIgnoringElementContentWhitespace(true);
47
48             DocumentBuilder builder = factory.newDocumentBuilder();
49             Document doc = builder.parse(file);
50             parseGridbag(doc.getDocumentElement());
51         }
52         catch (Exception e)
53         {
54             e.printStackTrace();
55         }
56     }
57
58     /**
59
```

```
59     * Получает компонент по заданному имени
60     * @param name Имя компонента
61     * @return Возвращает компонент с заданным именем или пустое
62     *         значение null, если в данной компоновке панели
63     *         отсутствует компонент с заданным именем
64 */
65 public Component get(String name)
66 {
67     Component[] components = getComponents();
68     for (int i = 0; i < components.length; i++)
69     {
70         if (components[i].getName().equals(name))
71             return components[i];
72     }
73     return null;
74 }
75 /**
76     * Выполняет синтаксический анализ элемента
77     * сеточно-контейнерной компоновки
78     * @param e Элемент сеточно-контейнерной компоновки
79 */
80 private void parseGridbag(Element e)
81 {
82     NodeList rows = e.getChildNodes();
83     for (int i = 0; i < rows.getLength(); i++)
84     {
85         Element row = (Element) rows.item(i);
86         NodeList cells = row.getChildNodes();
87         for (int j = 0; j < cells.getLength(); j++)
88         {
89             Element cell = (Element) cells.item(j);
90             parseCell(cell, i, j);
91         }
92     }
93 }
94 /**
95     * Выполняет синтаксический анализ элемента ячейки
96     * @param e Элемент ячейки
97     * @param r Ряд ячейки
98     * @param c Столбец ячейки
99 */
100 private void parseCell(Element e, int r, int c)
101 {
102     // получить атрибуты
103
104     String value = e.getAttribute("gridx");
105     if (value.length() == 0) // использовать по умолчанию
106     {
107         if (c == 0) constraints.gridx = 0;
108         else constraints.gridx += constraints.gridwidth;
109     }
110     else constraints.gridx = Integer.parseInt(value);
111
112     value = e.getAttribute("gridy");
113     if (value.length() == 0) // использовать по умолчанию
114     constraints.gridy = r;
115     else constraints.gridy = Integer.parseInt(value);
116
117     constraints.gridwidth =
118 }
```

```
119     Integer.parseInt(e.getAttribute("gridwidth"));
120     constraints.gridheight =
121         Integer.parseInt(e.getAttribute("gridheight"));
122     constraints.weightx =
123         Integer.parseInt(e.getAttribute("weightx"));
124     constraints.weighty =
125         Integer.parseInt(e.getAttribute("weighty"));
126     constraints.ipadx =
127         Integer.parseInt(e.getAttribute("ipadx"));
128     constraints.ipady =
129         Integer.parseInt(e.getAttribute("ipady"));
130     // использовать отражение для получения целых значений
131     // из статических полей
132     Class<GridBagConstraints> cl = GridBagConstraints.class;
133
134     try
135     {
136         String name = e.getAttribute("fill");
137         Field f = cl.getField(name);
138         constraints.fill = f.getInt(cl);
139
140         name = e.getAttribute("anchor");
141         f = cl.getField(name);
142         constraints.anchor = f.getInt(cl);
143     }
144     catch (Exception ex) // методы рефлексии могут генерировать
145                         // различные исключения
146     {
147         ex.printStackTrace();
148     }
149
150     Component comp = (Component) parseBean((Element)
151                                         e.getFirstChild());
152     add(comp, constraints);
153 }
154
155 /**
156 * Выполняет синтаксический анализ элемента JavaBeans
157 * @param e а Элемент JavaBeans
158 */
159 private Object parseBean(Element e)
160 {
161     try
162     {
163         NodeList children = e.getChildNodes();
164         Element classElement = (Element) children.item(0);
165         String className =
166             ((Text) classElement.getFirstChild()).getData();
167
168         Class<?> cl = Class.forName(className);
169
170         Object obj = cl.newInstance();
171
172         if (obj instanceof Component)
173             ((Component) obj).setName(e.getAttribute("id"));
174
175         for (int i = 1; i < children.getLength(); i++)
176         {
177             Node propertyElement = children.item(i);
178             Element nameElement =
```

```

179         (Element) propertyElement.getFirstChild();
180     String propertyName =
181         ((Text) nameElement.getFirstChild()).getData();
182
183     Element valueElement =
184         (Element) propertyElement.getLastChild();
185     Object value = parseValue(valueElement);
186     BeanInfo beanInfo = Introspector.getBeanInfo(cl);
187     PropertyDescriptor[] descriptors =
188         beanInfo.getPropertyDescriptors();
189     boolean done = false;
190     for (int j = 0; !done && j < descriptors.length; j++)
191     {
192         if (descriptors[j].getName().equals(propertyName))
193         {
194             descriptors[j].getWriteMethod().invoke(obj, value);
195             done = true;
196         }
197     }
198 }
199     return obj;
200 }
201 catch (Exception ex) // методы рефлексии могут генерировать
202 // различные исключения
203 {
204     ex.printStackTrace();
205     return null;
206 }
207 }
208
209 /**
210 * Выполняет синтаксический анализ элемента значения
211 * @param e Элемент значения
212 */
213 private Object parseValue(Element e)
214 {
215     Element child = (Element) e.getFirstChild();
216     if (child.getTagName().equals("bean"))
217         return parseBean(child);
218     String text = ((Text) child.getFirstChild()).getData();
219     if (child.getTagName().equals("int"))
220         return new Integer(text);
221     else if (child.getTagName().equals("boolean"))
222         return new Boolean(text);
223     else if (child.getTagName().equals("string")) return text;
224     else return null;
225 }
226 }
```

Листинг 3.4. Исходный код из файла read/fontdialog.xml

```

1  <?xml version="1.0"?>
2  <!DOCTYPE gridbag SYSTEM "gridbag.dtd">
3  <gridbag>
4      <row>
5          <cell anchor="EAST">
6              <bean>
7                  <class>javax.swing.JLabel</class>
8                  <property>
```

```
9          <name>text</name>
10         <value><string>Face: </string></value>
11     </property>
12   </bean>
13 </cell>
14 <cell fill="HORIZONTAL" weightx="100">
15   <bean id="face">
16     <class>javax.swing.JComboBox</class>
17   </bean>
18 </cell>
19 <cell gridheight="4" fill="BOTH" weightx="100" weighty="100">
20   <bean id="sample">
21     <class>javax.swing.JTextArea</class>
22   <property>
23     <name>text</name>
24     <value><string>
25       The quick brown fox jumps over the lazy dog
26     </string></value>
27   </property>
28   <property>
29     <name>editable</name>
30     <value><boolean>false</boolean></value>
31   </property>
32   <property>
33     <name>rows</name>
34     <value><int>8</int></value>
35   </property>
36   <property>
37     <name>columns</name>
38     <value><int>20</int></value>
39   </property>
40   <property>
41     <name>lineWrap</name>
42     <value><boolean>true</boolean></value>
43   </property>
44   <property>
45     <name>border</name>
46     <value>
47       <bean>
48         <class>javax.swing.border.EtchedBorder</class>
49       </bean>
50     </value>
51   </property>
52 </bean> .
53 </cell>
54 </row>
55 <row>
56 <cell anchor="EAST">
57   <bean>
58     <class>javax.swing.JLabel</class>
59   <property>
60     <name>text</name>
61     <value><string>Size: </string></value>
62   </property>
63 </bean>
64 </cell>
65 <cell fill="HORIZONTAL" weightx="100">
66   <bean id="size">
67     <class>javax.swing.JComboBox</class>
68   </bean>
69 </cell>
```

```

70      </row>
71      <row>
72          <cell gridwidth="2" weighty="100">
73              <bean id="bold">
74                  <class>javax.swing.JCheckBox</class>
75                  <property>
76                      <name>text</name>
77                      <value><string>Bold</string></value>
78                  </property>
79              </bean>
80          </cell>
81      </row>
82      <row>
83          <cell gridwidth="2" weighty="100">
84              <bean id="italic">
85                  <class>javax.swing.JCheckBox</class>
86                  <property>
87                      <name>text</name>
88                      <value><string>Italic</string></value>
89                  </property>
90              </bean>
91          </cell>
92      </row>
93  </gridbag>

```

Листинг 3.5. Исходный код из файла read/gridbag.dtd

```

1  <!ELEMENT gridbag (row*)>
2  <!ELEMENT row (cell*)>
3  <!ELEMENT cell (bean)>
4  <!ATTLIST cell gridx CDATA #IMPLIED>
5  <!ATTLIST cell gridy CDATA #IMPLIED>
6  <!ATTLIST cell gridwidth CDATA "1">
7  <!ATTLIST cell gridheight CDATA "1">
8  <!ATTLIST cell weightx CDATA "0">
9  <!ATTLIST cell weighty CDATA "0">
10 <!ATTLIST cell fill (NONE|BOTH|HORIZONTAL|VERTICAL) "NONE">
11 <!ATTLIST cell anchor
12     (CENTER|NORTH|NORTHEAST|EAST|SOUTHEAST|SOUTH|SOUTHWEST|
13     WEST|NORTHWEST) "CENTER">
14 <!ATTLIST cell ipadx CDATA "0">
15 <!ATTLIST cell ipady CDATA "0">
16
17 <!ELEMENT bean (class, property*)>
18 <!ATTLIST bean id ID #IMPLIED>
19
20 <!ELEMENT class (#PCDATA)>
21 <!ELEMENT property (name, value)>
22 <!ELEMENT name (#PCDATA)>
23 <!ELEMENT value (int|string|boolean|bean)>
24 <!ELEMENT int (#PCDATA)>
25 <!ELEMENT string (#PCDATA)>
26 <!ELEMENT boolean (#PCDATA)>

```

Листинг 3.6. Исходный код из файла read/gridbag.xsd

```

1 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
2
3     <xsd:element name="gridbag" type="GridBagType"/>

```

```
4      <xsd:element name="bean" type="BeanType"/>
5
6      <xsd:complexType name="GridBagType">
7          <xsd:sequence>
8              <xsd:element name= "row"
9                  type="RowType" minOccurs="0" maxOccurs="unbounded"/>
10             </xsd:sequence>
11         </xsd:complexType>
12
13     <xsd:complexType name="RowType">
14         <xsd:sequence>
15             <xsd:element name= "cell"
16                 type="CellType" minOccurs="0" maxOccurs="unbounded"/>
17             </xsd:sequence>
18         </xsd:complexType>
19
20     <xsd:complexType name="CellType">
21         <xsd:sequence>
22             <xsd:element ref="bean"/>
23         </xsd:sequence>
24         <xsd:attribute name="gridx" type="xsd:int" use="optional"/>
25         <xsd:attribute name="gridy" type="xsd:int" use="optional"/>
26         <xsd:attribute name="gridwidth"
27             type="xsd:int" use="optional" default="1" />
28         <xsd:attribute name="gridheight"
29             type="xsd:int" use="optional" default="1" />
30         <xsd:attribute name="weightx"
31             type="xsd:int" use="optional" default="0" />
32         <xsd:attribute name="weighty"
33             type="xsd:int" use="optional" default="0" />
34         <xsd:attribute name="fill" use="optional" default="NONE">
35             <xsd:simpleType>
36                 <xsd:restriction base="xsd:string">
37                     <xsd:enumeration value="NONE" />
38                     <xsd:enumeration value="BOTH" />
39                     <xsd:enumeration value="HORIZONTAL" />
40                     <xsd:enumeration value="VERTICAL" />
41                 </xsd:restriction>
42             </xsd:simpleType>
43         </xsd:attribute>
44     <xsd:attribute name="anchor" use="optional" default="CENTER">
45         <xsd:simpleType>
46             <xsd:restriction base="xsd:string">
47                 <xsd:enumeration value="CENTER" />
48                 <xsd:enumeration value="NORTH" />
49                 <xsd:enumeration value="NORTHEAST" />
50                 <xsd:enumeration value="EAST" />
51                 <xsd:enumeration value="SOUTHEAST" />
52                 <xsd:enumeration value="SOUTH" />
53                 <xsd:enumeration value="SOUTHWEST" />
54                 <xsd:enumeration value="WEST" />
55                 <xsd:enumeration value="NORTHWEST" />
56             </xsd:restriction>
57         </xsd:simpleType>
58     </xsd:attribute>
59     <xsd:attribute name="ipady"
60         type="xsd:int" use="optional" default="0" />
61     <xsd:attribute name="ipadx"
62         type="xsd:int" use="optional" default="0" />
63 </xsd:complexType>
```

```

65  <xsd:complexType name="BeanType">
66      <xsd:sequence>
67          <xsd:element name="class" type="xsd:string"/>
68          <xsd:element name="property"
69              type="PropertyType" minOccurs="0" maxOccurs="unbounded"/>
70      </xsd:sequence>
71      <xsd:attribute name="id" type="xsd:ID" use="optional" />
72  </xsd:complexType>
73
74  <xsd:complexType name="PropertyType">
75      <xsd:sequence>
76          <xsd:element name="name" type="xsd:string"/>
77          <xsd:element name="value" type="ValueType"/>
78      </xsd:sequence>
79  </xsd:complexType>
80
81  <xsd:complexType name="ValueType">
82      <xsd:choice>
83          <xsd:element ref="bean"/>
84          <xsd:element name="int" type="xsd:int"/>
85          <xsd:element name="string" type="xsd:string"/>
86          <xsd:element name="boolean" type="xsd:boolean"/>
87      </xsd:choice>
88  </xsd:complexType>
89 </xsd:schema>
90

```

3.4. Поиск информации средствами XPath

Если требуется найти информацию в XML-документе, придется организовать обход дерева модели DOM. Язык XPath упрощает доступ к узлам дерева. Допустим, имеется следующий XML-документ:

```

<configuration>
    ...
    <database>
        <username>dbuser</username>
        <password>secret</password>
    ...
</database>
</configuration>

```

Чтобы получить из него имя пользователя базы данных, достаточно вычислить следующее выражение XPath:

```
/configuration/database/username
```

Сделать это намного проще, чем организовывать непосредственный обход дерева модели DOM, выполнив перечисленные ниже действия.

1. Получить узел документа.
2. Перечислить его дочерние узлы.
3. Обнаружить элемент разметки database.
4. Получить его первый дочерний элемент username.
5. Получить его первый дочерний узел text.
6. Получить из него данные.

Язык XPath позволяет описывать ряд узлов в XML-документе. Например, в следующем выражении описывается ряд элементов `row`, которые являются дочерними для корневого элемента разметки `gridbag`:

```
/gridbag/row
```

Для выбора конкретного элемента служит операция `[]`. Так, в следующем выражении определяется первый дочерний элемент разметки (отчет индексов начинается с единицы):

```
/gridbag/row[1]
```

Для получения значений атрибутов служит операция `@`. Например, в следующем выражении XPath описывается атрибут `anchor` первой ячейки в первом ряду:

```
/gridbag/row[1]/cell[1]/@anchor
```

И наконец, в приведенном ниже выражении XPath описываются все узлы атрибутов `anchor` для элементов разметки `cell`, которые располагаются в элементах разметки `row`, а те, в свою очередь, являются дочерними для корневого элемента разметки `gridbag`.

```
/gridbag/row/cell/@anchor
```

В языке XPath имеется ряд функций, упрощающих работу с документом. Например, в следующем выражении определяется количество элементов `row`, дочерних для корневого элемента `gridbag`:

```
count (/gridbag/row)
```

Примеры выражений XPath, в том числе и довольно сложных, можно найти в спецификации этого языка по адресу <http://www.w3c.org/TR/xpath>. Имеется также очень удачно составленное руководство по XPath, доступное по адресу http://www.zvon.org/xxl/XPathTutorial/General_rus/examples.html.

В версии Java SE 5.0 внедрен прикладной программный интерфейс API для обработки выражений XPath. Сначала средствами класса `XPathFactory` создается объект типа `XPath`:

```
XPathFactory xpfactory = XPathFactory.newInstance();
path = xpfactory.newXPath();
```

Затем вызывается приведенный ниже метод `evaluate()`, обрабатывающий выражения XPath. С помощью одного объекта типа `XPath` можно обработать несколько выражений.

```
String username =
    path.evaluate("/configuration/database/username", doc);
```

В данной форме метод `evaluate()` возвращает результат в виде символьной строки. Это удобно для получения текста, например, из узла `username`, как показано выше. Если из выражения XPath получается ряд узлов, то для их обработки можно сделать следующий вызов:

```
NodeList nodes = (NodeList) path.evaluate(
    "/gridbag/row", doc, XPathConstants.NODESET);
```

Если же в итоге получается один узел, то в качестве третьего параметра при вызове метода `evaluate()` следует указать значение константы `XPathConstants.NODE`:

```
Node node = (Node) path.evaluate(
    "/gridbag/row[1]", doc, XPathConstants.NODE);
```

А если в итоге получается количество узлов, то в качестве третьего параметра при вызове метода evaluate() следует указать значение константы XPathConstants.NUMBER:

```
int count = ((Number) path.evaluate("count(/gridbag/row)", doc,
    XPathConstants.NUMBER)).intValue();
```

Поиск совсем не обязательно начинать с корневого узла документа. В качестве исходной точки можно выбрать любой узел и даже перечень узлов. Например, получив узел в результате вычисления приведенного выше выражения, можно сделать следующий вызов:

```
result = path.evaluate(expression, node);
```

В примере программы, исходный код которой приведен в листинге 3.7, демонстрируется порядок вычисления выражений XPath. Загрузите сначала XML-файл и введите выражение. Затем выберите тип выражения и щелкните на кнопке Evaluate (Вычислить). Результат вычисления введенного выражения появится в нижней части окна (рис. 3.5).

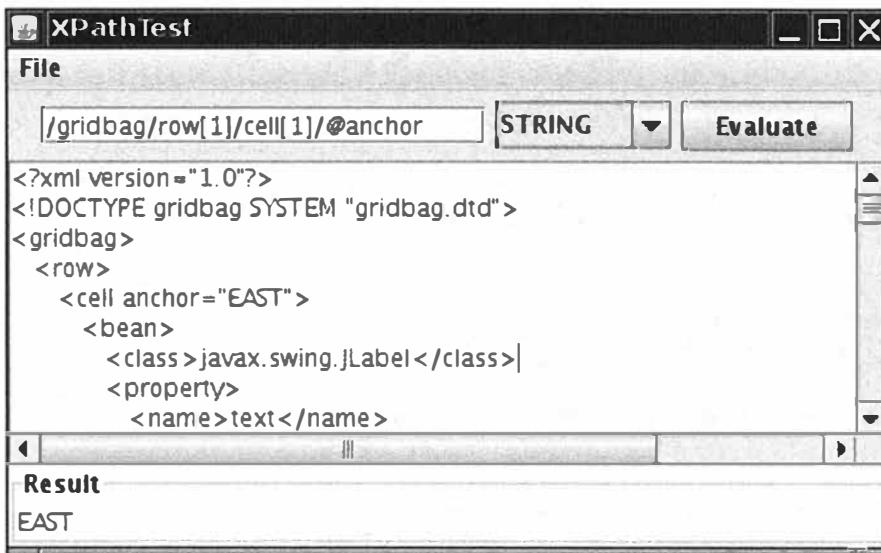


Рис. 3.5. Вычисление выражений XPath

Листинг 3.7. Исходный код из файла xpath/XPathTester.java

```
1 package xpath;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.io.*;
6 import java.nio.file.*;
7 import java.util.*;
```

```
8  import javax.swing.*;
9  import javax.swing.border.*;
10 import javax.xml.namespace.*;
11 import javax.xml.parsers.*;
12 import javax.xml.xpath.*;
13 import org.w3c.dom.*;
14 import org.xml.sax.*;
15
16 /**
17 * В этой программе вычисляются выражения XPath
18 * @version 1.02 2016-05-10
19 * @author Cay Horstmann
20 */
21 public class XPathTester
22 {
23     public static void main(String[] args)
24     {
25         EventQueue.invokeLater(() ->
26         {
27             JFrame frame = new XPathFrame();
28             frame.setTitle("XPathTest");
29             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30             frame.setVisible(true);
31         });
32     }
33 }
34
35 /**
36 * В этом фрейме отображается XML-документ, панель для ввода
37 * выражений XPath, а также текстовое поле для вывода результата
38 * display the result.
39 */
40 class XPathFrame extends JFrame
41 {
42     private DocumentBuilder builder;
43     private Document doc;
44     private XPath path;
45     private JTextField expression;
46     private JTextField result;
47     private JTextArea docText;
48     private JComboBox<String> typeCombo;
49
50     public XPathFrame()
51     {
52         JMenu fileMenu = new JMenu("File");
53         JMenuItem openItem = new JMenuItem("Open");
54         openItem.addActionListener(event -> openFileDialog());
55         fileMenu.add(openItem);
56
57         JMenuItem exitItem = new JMenuItem("Exit");
58         exitItem.addActionListener(event -> System.exit(0));
59         fileMenu.add(exitItem);
60
61         JMenuBar menuBar = new JMenuBar();
62         menuBar.add(fileMenu);
63         setJMenuBar(menuBar);
64
65         ActionListener listener = event -> evaluate();
66         expression = new JTextField(20);
67         expression.addActionListener(listener);
```

```
68     JButton evaluateButton = new JButton("Evaluate");
69     evaluateButton.addActionListener(listener);
70
71     typeCombo = new JComboBox<String>(new String[] {
72         "STRING", "NODE", "NODESET", "NUMBER", "BOOLEAN" });
73     typeCombo.setSelectedItem("STRING");
74
75     JPanel panel = new JPanel();
76     panel.add(expression);
77     panel.add(typeCombo);
78     panel.add(evaluateButton);
79     docText = new JTextArea(10, 40);
80     result = new JTextField();
81     result.setBorder(new TitledBorder("Result"));
82
83     add(panel, BorderLayout.NORTH);
84     add(new JScrollPane(docText), BorderLayout.CENTER);
85     add(result, BorderLayout.SOUTH);
86
87     try
88     {
89         DocumentBuilderFactory factory =
90             DocumentBuilderFactory.newInstance();
91         builder = factory.newDocumentBuilder();
92     }
93     catch (ParserConfigurationException e)
94     {
95         JOptionPane.showMessageDialog(this, e);
96     }
97
98     XPathFactory xpfactory = XPathFactory.newInstance();
99     path = xpfactory.newXPath();
100    pack();
101 }
102
103 /**
104 * Открыть файл и загрузить документ
105 */
106 public void openFile()
107 {
108     JFileChooser chooser = new JFileChooser();
109     chooser.setCurrentDirectory(new File("xpath"));
110
111     chooser.setFileFilter(
112         new javax.swing.filechooser.FileNameExtensionFilter(
113             "XML files", "xml"));
114     int r = chooser.showOpenDialog(this);
115     if (r != JFileChooser.APPROVE_OPTION) return;
116     File file = chooser.getSelectedFile();
117     try
118     {
119         docText.setText(new String(
120             Files.readAllBytes(file.toPath())));
121         doc = builder.parse(file);
122     }
123     catch (IOException e)
124     {
125         JOptionPane.showMessageDialog(this, e);
126     }
127     catch (SAXException e)
```

```

128     {
129         JOptionPane.showMessageDialog(this, e);
130     }
131 }
132
133 public void evaluate()
134 {
135     try
136     {
137         String typeName = (String) typeCombo.getSelectedItem();
138         QName returnType = (QName)
139             XPathConstants.class.getField(typeName).get(null);
140         Object evalResult = path.evaluate(expression.getText(),
141             doc, returnType);
142         if (typeName.equals("NODESET"))
143         {
144             NodeList list = (NodeList) evalResult;
145             // Нельзя воспользоваться методом String.join(),
146             // т.к. объект типа NodeList не является итерируемым
147             StringJoiner joiner = new StringJoiner(", ", "{", "}");
148             for (int i = 0; i < list.getLength(); i++)
149                 joiner.add("") + list.item(i));
150             result.setText("") + joiner);
151         }
152         else result.setText("") + evalResult);
153     }
154     catch (XPathExpressionException e)
155     {
156         result.setText("") + e);
157     }
158     catch (Exception e) // исключение при рефлексии
159     {
160         e.printStackTrace();
161     }
162 }
163 }

```

javax.xml.xpath.XPathFactory 5.0

- **static XPathFactory newInstance()**

Возвращает экземпляр класса **XPathFactory**, используемый для создания объектов типа **XPath**.

- **XPath newPath()**

Создает объект типа **XPath**, который можно использовать для обработки выражений **XPath**.

javax.xml.xpath.XPath 5.0

- **String evaluate(String expression, Object startingPoint)**

Вычисляет выражение, начиная поиск с заданной исходной точки. В качестве исходной точки может быть указан узел или перечень узлов. Если в результате вычисления данного выражения получается узел или ряд узлов, то возвращаемая символьная строка содержит данные из всех дочерних текстовых узлов.

javax.xml.xpath.XPath 5.0 (окончание)

- **Object evaluate(String expression, Object startingPoint, QName resultType)**

Вычисляет выражение, начиная поиск с заданной исходной точки. В качестве исходной точки может быть указан узел или перечень узлов. В качестве параметра **resultType** задается одна из следующих констант, определяемых в классе **XPathConstants**: **STRING**, **NODE**, **NODESET**, **NUMBER** или **BOOLEAN**. Возвращаемое значение относится к типу **String**, **Node**, **NodeList**, **Number** или **Boolean**.

3.5. Использование пространств имен

Во избежание конфликтов при использовании одинаковых имен в языке Java предусмотрены пакеты. Разные классы могут иметь одинаковые имена, если они находятся в разных пакетах. А в XML для различия одинаково именуемых элементов и атрибутов используется механизм *пространств имен*. Пространство имен обозначается с помощью универсального идентификатора ресурсов (URI), как показано в приведенном ниже примере.

<http://www.w3.org/2001/XMLSchema>
uuid:1c759aed-b748-475c-ab68-10679700c4f2
urn:com:books-r-us

Чаще всего для этой цели используется формат URL по сетевому протоколу HTTP. Однако URL в данном случае выполняет лишь роль идентификатора. Приведенные ниже URL обозначают *разные* пространства имен, хотя веб-сервер интерпретировал бы их как указатели на один и тот же документ.

<http://www.horstmann.com/corejava>
<http://www.horstmann.com/corejava/index.html>

Более того, URL, определяющий пространство имен, может не указывать на конкретный документ. XML-анализатор и не пытается найти что-нибудь по этому адресу. Тем не менее по URL, обозначающему пространство имен, принято располагать документ с описанием назначения этого пространства. Например, по адресу <http://www.w3c.org/2001/XMLSchema> находится документ с описанием стандарта XML Schema.

А зачем для обозначения пространства имен применяются URL? Очевидно, что в таком случае легче гарантировать их однозначность. В самом деле, для подлинного URL однозначность имени узла сети сети гарантируется структурой системы доменных имен, а однозначность остальной части URL должна обеспечить надлежащая организация файловой системы. Именно из этих соображений для многих пакетов выбраны доменные имена с обратным порядком следования доменов.

Безусловно, для длинных имен проще обеспечить однозначность, но использовать их в программе не совсем удобно. В языке Java для этой цели предусмотрен механизм импорта пакетов, в результате чего в исходном тексте программы присутствуют в основном имена классов. Аналогичный механизм предусмотрен и в XML, как показано ниже. В итоге элемент и все его дочерние узлы становятся частью заданного пространства имен.

```
<элемент xmlns="URI_пространства_имен">
    дочерние_узлы
</элемент>
```

При необходимости дочерний узел может обеспечить себе отдельное пространство имен, как показано в приведенном ниже примере. В данном случае первый дочерний узел и дочерние узлы предыдущего уровня принадлежат второму пространству имен.

```
<элемент xmlns="URI_пространства_имен_1">
    <дочерний_узел xmlns="URI_пространства_имен_2">
        дочерние_узлы_предыдущего_уровня
    </дочерний_узел>
        другие дочерние_узлы
</элемент>
```

Этот простой механизм подходит только в том случае, если требуется одно пространство имен или же если пространства имен вложены друг в друга естественным образом. В иных случаях более предпочтительным оказывается альтернативный механизм, аналог которого отсутствует в Java. Для пространства имен можно использовать *префикс*, т.е. короткий идентификатор, выбираемый для конкретного документа. Ниже приведен типичный пример применения префикса *xsd* в файле схемы типа XML Schema.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="gridbag" type="GridBagType"/>
    .
    .
</xsd:schema>
```

Атрибут *xmlns:префикс="URI_пространства_имен"* определяет пространство имен и префикс. В данном примере префиксом является символьная строка "xsd". Таким образом, атрибут *xsd: schema* фактически означает следующее: указанная схема (*schema*) находится в пространстве имен <http://www.w3.org/2001/XMLSchema>.



На заметку! Пространство имен родительского элемента наследуется только дочерними элементами. Атрибуты без явного указания префикса не считаются частью пространства имен. Рассмотрим следующий вымышленный пример:

```
<configuration xmlns="http://www.horstmann.com/corejava"
    xmlns:si="http://www.bipm.fr/enus/3_SI/si.html">
    <size value="210" si:unit="mm"/>
    .
    .
</configuration>
```

В данном примере элементы разметки *configuration* и *size* являются частью пространства имен, определяемого по следующему URI: <http://www.horstmann.com/corejava>. Атрибут *si:unit* является частью пространства имен по следующему URI: http://www.bipm.fr/enus/3_SI/si.html. Но атрибут *value* не принадлежит ни одному из этих пространств имен.

Манипулирование пространствами имен в синтаксическом анализаторе поддается контролю. По умолчанию в DOM-анализаторе пространства имен во внимание не принимаются. Чтобы включить режим управления

пространствами имен, достаточно вызвать метод `setNamespaceAware()` из класса `DocumentBuilderFactory` следующим образом:

```
factory.setNamespaceAware(true);
```

После этого все созданные данной фабрикой конструкторы будут поддерживать пространства имен. У каждого узла имеются следующие три свойства.

- **Уточненное имя** с префиксом, возвращаемое методами `getnodeName()`, `getTagName()` и т.д.
- **URI пространства имен**, возвращаемый методом `getNamespaceURI()`.
- **Локальное имя** без префикса, возвращаемое методом `getLocalName()`.

Обратимся к конкретному примеру. Допустим, синтаксический анализатор обнаруживает следующий элемент разметки:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

В таком случае он сообщает о наличии следующих свойств узла.

- Уточненное имя: `xsd:schema`.
- URI пространства имен: `http://www.w3.org/2001/XMLSchema`.
- Локальное имя: `schema`.



На заметку! Если режим управления пространствами имен отключен, методы `getNamespaceURI()` и `getLocalName()` возвращают пустое значение `null`.

org.w3c.dom.Node 1.4

- **String getLocalName()**

Возвращает локальное имя (без префикса) или пустое значение `null`, если режим управления пространствами имен отключен.

- **String getNamespaceURI()**

Возвращает URI пространства имен или пустое значение `null`, если узел не является частью пространства имен или режим управления пространствами имен отключен.

javax.xml.parsers.DocumentBuilderFactory 1.4

- **boolean isNamespaceAware()**
- **void setNamespaceAware(boolean value)**

Получают или устанавливают значение свойства, определяющего режим управления пространствами имен. Если установлено логическое значение `true` параметра `value`, то в генерируемых фабрикой синтаксических анализаторах будет включен режим управления пространствами имен.

3.6. Потоковые синтаксические анализаторы

DOM-анализатор считывает XML-документ и представляет его в виде деревовидной структуры данных. Для большинства приложений оказывается

достаточно и модели DOM. Но она неэффективна, если документ крупный, а алгоритм его обработки слишком прост, чтобы оперативно анализировать узлы, не просматривая дерево в целом. В подобных случаях следует применять потоковые синтаксические анализаторы.

В последующих разделах будут рассмотрены потоковые анализаторы, доступные в библиотеке Java: почтенный SAX-анализатор и более современный StAX-анализатор, который появился в версии Java SE 6. SAX-анализатор использует обратные вызовы событий, а StAX-анализатор предоставляет итератор событий синтаксического анализа. Последний оказывается более удобным в употреблении.

3.6.1. Применение SAX-анализатора

SAX-анализатор уведомляет о событиях, наступающих в ходе синтаксического анализа компонентов данных, вводимых из XML-документа. Но сам документ не хранится в памяти, а создание структуры из вводимых данных возлагается на обработчики событий. На самом деле в основу работы DOM-анализатора положен тот же самый принцип, что и для SAX-анализатора: дерево модели DOM строится по мере приема событий, наступающих при синтаксическом анализе.

Чтобы воспользоваться SAX-анализатором, нужно создать обработчик событий, определяющий действия для обработки различных событий, наступающих при синтаксическом анализе. В интерфейсе `ContentHandler` определен ряд перечисленных ниже методов обратного вызова, к которым анализатор обращается в ходе синтаксического анализа XML-документа.

- Методы `startElement()` и `endElement()` вызываются всякий раз, когда получается открывающий и закрывающий дескрипторы.
- Метод `characters()` вызывается при получении символьных данных.
- Методы `startDocument()` и `endDocument()` вызываются в начале и в конце документа.

Например, при синтаксическом анализе следующего фрагмента разметки:

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

SAX-анализатор генерирует обратные вызовы перечисленных ниже методов.

1. Метод `startElement()`, имя элемента разметки: `font`.
2. Метод `startElement()`, имя элемента разметки: `name`.
3. Метод `characters()`, содержимое: `Helvetica`.
4. Метод `endElement()`, имя элемента разметки: `name`.
5. Метод `startElement()`, имя элемента разметки: `size`, атрибуты: `units="pt"`.
6. Метод `characters()`, содержимое: `36`.
7. Метод `endElement()`, имя элемента разметки: `size`.
8. Метод `endElement()`, имя элемента разметки: `font`.

Для выполнения требуемых действий при синтаксическом анализе содержимого вводимого файла необходимо переопределить эти методы. В примере программы, исходный код которой приведен в листинге 3.8, выводятся сведения обо всех гипертекстовых ссылках типа ``, найденных в HTML-файле. В ней переопределяется метод `startElement()` обработчика событий и реализуется проверка всех гипертекстовых ссылок с именем `a` и атрибутом `href`. Подобный код применяется в поисковых роботах, которые автоматически выявляют новые веб-страницы для индексации, переходя по ссылкам.



На заметку! К сожалению, многие HTML-страницы совершенно не соответствуют формату XML, и поэтому рассматриваемая здесь программа не способна произвести их синтаксический анализ. Тем не менее большинство веб-страниц, созданных консорциумом W3C, составлены на XHTML (диалекте HTML, соответствующем формату XML). Поэтому этими веб-страницами можно воспользоваться для тестирования данной программы. Например, для составления списка всех гипертекстовых ссылок по соответствующим URL на веб-странице <http://www.w3c.org/MarkUp> необходимо ввести следующую команду:

```
java SAXTest http://www.w3c.org/MarkUp
```

Рассматриваемая здесь программа служит характерным примером употребления SAX-анализатора. В ней полностью игнорируется контекст, в котором находится элемент разметки `a`, а также не сохраняется древовидная структура документа. Для получения SAX-анализатора используется следующий фрагмент кода:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
```

После этого документ можно обработать следующим образом:

```
parser.parse(source, handler)
```

где `source` — источник входных данных, который может быть файлом, символьной строкой с URL или потоком ввода, а `handler` — подкласс, производный от класса `DefaultHandler`. В классе `DefaultHandler` определены методы, объявленные в следующих интерфейсах:

```
ContentHandler
DTDHandler
EntityResolver
ErrorHandler
```

Эти методы не выполняют никаких действий. В рассматриваемой здесь программе определен обработчик событий, в котором для поиска элементов с именем `a` и атрибутом `href` переопределяется метод `startElement()` из интерфейса `ContentHandler`:

```
DefaultHandler handler = new DefaultHandler()
{
    public void startElement(String namespaceURI, String lname,
                            String qname, Attributes attrs)
    {
        if (lname.equalsIgnoreCase("a") && attrs != null)
        {
            for (int i = 0; i < attrs.getLength(); i++)
            {
                String aname = attrs.getLocalName(i);
```

```
        if (aname.equalsIgnoreCase("href"))
            System.out.println(attrs.getValue(i));
    }
}
};
```

Методы `startElement()` передаются три параметра, описывающие имя элемента. В частности, параметр `qName` сообщает уточненное имя в форме префикса:локальное_имя. Если включен режим управления пространствами имен, то параметры `namespaceURI` и `lName` описывают пространство имен и локальное (неуточненное) имя.

Как и при использовании DOM-анализатора, режим управления пространствами имен исходно отключен. Для его включения достаточно вызвать метод `setNamespaceAware()` из фабричного класса `SAXParserFactory` следующим образом:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setNamespaceAware(true);
SAXParser saxParser = factory.newSAXParser();
```

В рассматриваемой здесь программе преодолевается еще одно распространенное препятствие. В начале XHTML-файла обычно находится дескриптор, содержащий ссылку на описание DTD, которое требуется загрузить синтаксическому анализатору. Очевидно, что консорциуму W3C явно не улыбалась перспектива обслуживать миллиарды копий файлов вроде www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd. Поэтому они вообще отказались от такого обслуживания, но на момент написания этой книги описание DTD все же обслуживалось, хотя и очень медленно. Если же вам не требуется проверка достоверности документа, просто сделайте следующий вызов:

```
factory.setFeature(  
    "http://apache.org/xml/features/nonvalidating/load-external-dtd",  
    false);
```

Итак, в листинге 3.8 приведен исходный код простейшего поискового робота. Далее в этой главе рассматривается еще один интересный пример применения SAX-анализатора. Чтобы превратить источник данных, несовместимый с форматом XML, в XML-документ, проще всего уведомить о SAX-событиях, о которых будет затем сообщать сам XML-анализатор. Более подробно эти вопросы рассматриваются далее, в разделе 3.8.

Листинг 3.8. Исходный код из файла sax/SAXTest.java

```
1 package sax;
2
3 import java.io.*;
4 import java.net.*;
5 import javax.xml.parsers.*;
6 import org.xml.sax.*;
7 import org.xml.sax.helpers.*;
8
9 /**
10 * В этой программе демонстрируется применение SAX-анализатора.
```

```
11 * Программа выводит все гиперссылки из веб-страницы формата XHTML
12 * Использование: java SAXTest url
13 * @version 1.00 2001-09-29
14 * @author Cay Horstmann
15 */
16 public class SAXTest
17 {
18     public static void main(String[] args) throws Exception
19     {
20         String url;
21         if (args.length == 0)
22         {
23             url = "http://www.w3c.org";
24             System.out.println("Using " + url);
25         }
26         else url = args[0];
27
28         DefaultHandler handler = new DefaultHandler()
29         {
30             public void startElement(String namespaceURI,
31                           String lname, String qname, Attributes attrs)
32             {
33                 if (lname.equals("a") && attrs != null)
34                 {
35                     for (int i = 0; i < attrs.getLength(); i++)
36                     {
37                         String aname = attrs.getLocalName(i);
38                         if (aname.equals("href"))
39                             System.out.println(attrs.getValue(i));
40                     }
41                 }
42             }
43         };
44
45         SAXParserFactory factory = SAXParserFactory.newInstance();
46         factory.setNamespaceAware(true);
47         factory.setFeature("http://
48             apache.org/xml/features/nonvalidating/load-external-dtd",
49             false);
50         SAXParser saxParser = factory.newSAXParser();
51         InputStream in = new URL(url).openStream();
52         saxParser.parse(in, handler);
53     }
54 }
```

javax.xml.parsers.SAXParserFactory 1.4

- **static SAXParserFactory newInstance()**
Возвращает экземпляр класса **SAXParserFactory**.
- **SAXParser newSAXParser()**
Возвращает экземпляр класса **SAXParser**.

javax.xml.parsers.SAXParserFactory 1.4

- `boolean isNamespaceAware()`
 - `void setNamespaceAware(boolean value)`

Получают или устанавливают значение свойства `namespaceAware`, определяющего режим управления пространствами имен в фабрике. Если в этом свойстве установлено логическое значение `true`, то режим управления пространствами имен активизирован для синтаксических анализаторов, генерируемых фабрикой.

- `boolean isValidating()`
 - `void setValidating(boolean value)`

Получают или устанавливают значение свойства **validating**, определяющего режим проверки достоверности вводимых данных в фабрике. Если в этом свойстве установлено логическое значение **true**, то режим проверки достоверности вводимых данных активизирован для синтаксических анализаторов, генерируемых фабрикой.

`javax.xml.parsers.SAXParser 1.4`

- void parse(File f, DefaultHandler handler)
 - void parse(String url, DefaultHandler handler)
 - void parse(InputStream in, DefaultHandler handler)

Выполняют синтаксический анализ XML-документа, полученного из файла по указанному URL или из потока ввода, а также оповещают заданный обработчик о событиях, наступающих в ходе синтаксического анализа.

`org.xml.sax.ContentHandler 1.4`

- `void startDocument()`
 - `void endDocument()`
Вызываются в начале и в конце XML-документа соответственно.
 - `void startElement(String uri, String lname,
String qname, Attributes attr)`
 - `void endElement(String uri, String lname, String qname)`
 - Вызываются в начале и в конце элемента разметки соответственно.

Параметры: **uri** URI пространства имен [если включен режим управления пространствами имен]

lname Локальное имя без префикса (если включен режим управления пространствами имен)

粹粹	Имя элемента (если отключен режим управления пространствами имен) или уточненное имя с префиксом (если анализатор сообщает уточненные имена помимо локальных имен)
-----------	--

org.xml.sax.ContentHandler 1.4 (окончание)

- **void characters(char[] data, int start, int length)**

Вызывается при получении символьных данных.

Параметры:

data

Массив символьных данных

start

Индекс первого символа в массиве, составляющем часть сообщаемых символьных данных

length

Длина сообщаемой символьной строки

org.xml.sax.Attributes 1.4

- **int getLength()**

Возвращает количество атрибутов, хранящихся в коллекции атрибутов.

- **String getLocalName(int index)**

Возвращает локальное имя (без префикса) атрибута по указанному индексу или пустую символьную строку, если для синтаксического анализатора не включен режим управления пространствами имен.

- **String getURI(int index)**

Возвращает URI пространства имен для атрибута по указанному индексу или пустую символьную строку, если узел не является частью пространства имен или же если для синтаксического анализатора не включен режим управления пространствами имен.

- **String getQName(int index)**

Возвращает уточненное имя (с префиксом) атрибута по указанному индексу или пустую символьную строку, если уточненное имя не сообщено синтаксическим анализатором.

- **String getValue(int index)**

- **String getValue(String qname)**

- **String getValue(String uri, String lname)**

Возвращают значение атрибута по указанному индексу, уточненное имя или URI пространства имен вместе с локальным именем. Если такое значение отсутствует, то возвращается пустое значение **null**.

3.6.2. Применение StAX-анализатора

StAX-анализатор является "извлекающим" синтаксическим анализатором. Вместо того чтобы устанавливать обработчик событий, достаточно произвести перебор событий в следующем цикле:

```
InputStream in = url.openStream();
XMLInputFactory factory = XMLInputFactory.newInstance();
XMLStreamReader parser = factory.createXMLStreamReader(in);
while (parser.hasNext())
{
    int event = parser.next();
    вызвать методы синтаксического анализатора parser,
    чтобы получить подробные сведения о событии
}
```

Например, в ходе синтаксического анализа следующего фрагмента разметки:

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

синтаксический анализатор выдает перечисленные ниже события.

1. START_ELEMENT, имя элемента: font.
2. CHARACTERS, содержимое: пробел.
3. START_ELEMENT, имя элемента разметки: name.
4. CHARACTERS, содержимое: Helvetica.
5. END_ELEMENT, имя элемента разметки: name.
6. CHARACTERS, содержимое: пробел.
7. START_ELEMENT, имя элемента разметки: size.
8. CHARACTERS, содержимое: 36.
9. END_ELEMENT, имя элемента разметки: size.
10. CHARACTERS, содержимое: пробел.
11. END_ELEMENT, имя элемента разметки: font.

Чтобы проанализировать значения атрибутов, следует вызвать соответствующие методы из класса `XMLStreamReader`. Например, при вызове следующего метода получается атрибут `units` текущего элемента:

```
String units = parser.getAttributeValue(null, "units");
```

По умолчанию режим управления пространствами имен включен. Отключить его можно, видоизменив фабрику следующим образом:

```
XMLInputFactory factory = XMLInputFactory.newInstance();
factory.setProperty(XMLInputFactory.IS_NAMESPACE_AWARE, false);
```

В листинге 3.9 приведен исходный код программы поискового робота, реализованной вместе со StAX-анализатором. Нетрудно заметить, что исходный код этой программы намного проще, чем код аналогичной программы с SAX-анализатором, поскольку в данном случае не нужно организовывать обработку событий.

Листинг 3.9. Исходный код из файла `stax/StAXTest.java`

```
1 package stax;
2
3 import java.io.*;
4 import java.net.*;
5 import javax.xml.stream.*;
6
7 /**
8  * В этой программе демонстрируется применение StAX-анализатора.
9  * Программа выводит все гиперссылки из веб-страницы формата XHTML
10 * Использование: java StAXTest url
11 * @author Cay Horstmann
12 * @version 1.0 2007-06-23
13 */
```

```

14 public class StAXTest
15 {
16     public static void main(String[] args) throws Exception
17     {
18         String urlString;
19         if (args.length == 0)
20         {
21             urlString = "http://www.w3c.org";
22             System.out.println("Using " + urlString);
23         }
24         else urlString = args[0];
25         URL url = new URL(urlString);
26         InputStream in = url.openStream();
27         XMLInputFactory factory = XMLInputFactory.newInstance();
28         XMLStreamReader parser = factory.createXMLStreamReader(in);
29         while (parser.hasNext())
30         {
31             int event = parser.next();
32             if (event == XMLStreamConstants.START_ELEMENT)
33             {
34                 if (parser.getLocalName().equals("a"))
35                 {
36                     String href = parser.getAttributeValue(null, "href");
37                     if (href != null)
38                         System.out.println(href);
39                 }
40             }
41         }
42     }
43 }

```

`javax.xml.stream.XMLInputFactory` 6• **`static XMLInputFactory newInstance()`**

Возвращает экземпляр класса `XMLInputFactory`.

• **`void setProperty(String name, Object value)`**

Задает свойство для данной фабрики или генерирует исключение типа `IllegalArgumentException`, если свойство не поддерживается или не допускает установку заданного значения. В реализации Java SE поддерживаются следующие свойства, допускающие установку логических значений:

`"javax.xml.stream.isValidating"`

При логическом значении

`false` (по умолчанию)

документ не проверяется.

В спецификации это свойство не требуется

`"javax.xml.stream.isNamespaceAware"`

При логическом значении

`true` (по умолчанию)

обрабатываются

пространства имен.

В спецификации это свойство не требуется

javax.xml.stream.XMLInputFactory 6**"javax.xml.stream.isCoalescing"**

При логическом значении **false** (по умолчанию) соседние символы не объединяются

"javax.xml.stream.isReplacingEntityReferences"

При логическом значении **true** (по умолчанию) ссылки на сущности заменяются и сообщаются в виде символьных данных

"javax.xml.stream.isSupportingExternalEntities"

При логическом значении **true** (по умолчанию) разрешаются внешние сущности. В спецификации не определяется никаких значений этого свойства по умолчанию

"javax.xml.stream.supportDTD"

При логическом значении **true** (по умолчанию) об описаниях DTD сообщается как о событиях

- XMLStreamReader createXMLStreamReader(InputStream in)**
- XMLStreamReader createXMLStreamReader(InputStream in, String characterEncoding)**
- XMLStreamReader createXMLStreamReader(Reader in)**
- XMLStreamReader createXMLStreamReader(Source in)**

Создают синтаксический анализатор, читающий данные из заданного потока ввода, потока чтения или источника JAXP.

javax.xml.stream.XMLStreamReader 6

- boolean hasNext()**

Возвращает логическое значение **true**, если существует другое событие синтаксического анализа.

- int next()**

Задает состояние синтаксического анализатора для последующего события синтаксического анализа и возвращает одну из следующих констант: **START_ELEMENT**, **END_ELEMENT**, **CHARACTERS**, **START_DOCUMENT**, **END_DOCUMENT**, **CDATA**, **COMMENT**, **SPACE** (игнорируемый пробел), **PROCESSING_INSTRUCTION**, **ENTITY_REFERENCE**, **DTD**.

javax.xml.stream.XMLStreamReader 6 (окончание)

- **boolean isStartElement()**
- **boolean isEndElement()**
- **boolean isCharacters()**
- **boolean isWhiteSpace()**

Возвращают логическое значение **true**, если текущее событие связано с начальным элементом, конечным элементом, символьными данными или разделителем в виде пробела.

- **QName getName()**
- **String getLocalName()**

Получают имя элемента в событии **START_ELEMENT** или **END_ELEMENT**.

- **String getText()**

Возвращают символы события **CHARACTERS**, **COMMENT** или **CDATA**, замещающее значение для константы **ENTITY_REFERENCE** или внутреннее подмножество DTD.

- **int getAttributeCount()**
- **QName getAttributeName(int index)**
- **StringgetAttributeLocalName(int index)**
- **StringgetAttributeValue(int index)**

Получают подсчет количества атрибутов, а также имена и значения атрибутов, при условии, что текущим оказывается событие **START_ELEMENT**.

- **StringgetAttributeValue(String namespaceURI, String name)**

Получает значение атрибута по данному имени, при условии, что текущим оказывается событие **START_ELEMENT**. Если параметр **namespaceURI** принимает пустое значение **null**, пространство имен не проверяется.

3.7. Формирование XML-документов

Итак, рассмотрев способы написания программ на Java, предназначенных для чтения XML-документов, перейдем к способам написания программ, выполняющих обратное действие, т.е. формирующих данные для вывода в формате XML. Разумеется, XML-документ можно сформировать, введя в программу последовательность вызовов метода **print()** и выводя с их помощью элементы разметки, атрибуты и текст. Но для создания такого громоздкого кода потребуется много времени и труда. Кроме того, подобный код, как правило, изобилует ошибками. Очень легко, например, ошибиться при употреблении специальных знаков " или < в значениях атрибутов и в тексте.

Намного удобнее создать дерево модели DOM, представляющей документ, а затем записать его содержимое в файл. Подробности такого подхода к формированию XML-документов обсуждаются в последующих разделах.

3.7.1. XML-документы без пространств имен

Чтобы построить древовидную структуру DOM, нужно сначала сформировать пустой документ с помощью метода **newDocument()** из класса **DocumentBuilder** следующим образом:

```
Document doc = builder.newDocument();
```

Затем следует вызвать метод `createElement()` из класса `Document`, чтобы построить элементы документа, как показано ниже.

```
Element rootElement = doc.createElement(rootName);
Element childElement = doc.createElement(childName);
```

Далее создаются текстовые узлы с помощью метода `createTextNode()`:

```
Text textNode = doc.createTextNode(textContents);
```

3.7.2. XML-документы с пространствами имен

Если используются пространства имен, то процедура формирования XML-документа будет несколько иной. Сначала фабрика построителей документов устанавливается в режим управления пространствами имен, а затем создается построитель документов, как показано ниже.

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setNamespaceAware(true);
builder = factory.newDocumentBuilder();
```

Далее для создания любых узлов вместо метода `createElement()` вызывается метод `createElementNS()`:

```
String namespace = "http://www.w3.org/2000/svg";
Element rootElement = doc.createElementNS(namespace, "svg");
```

Если узел имеет уточненное имя с префиксом пространства имен, то любые требующиеся атрибуты с префиксом `xmlns` создаются автоматически. Так, если требуется ввести данные формата SVG в XHTML-документ, для этой цели можно создать соответствующий элемент аналогично приведенному ниже.

```
Element svgElement = doc.createElement(namespace, "svg:svg")
```

А когда этот элемент записывается, он превращается в следующий элемент разметки:

```
<svg:svg xmlns:svg="http://www.w3.org/2000/svg">
```

Если же требуется установить атрибуты элемента разметки, имена которых находятся в отдельном пространстве имен, для этой цели вызывается метод `setAttributeNS()` из класса `Element`:

```
rootElement.setAttributeNS(namespace, qualifiedName, value);
```

3.7.3. Запись XML-документов

Как ни странно, записать дерево модели DOM в поток вывода не так-то просто. Для этой цели проще всего воспользоваться прикладным программным интерфейсом API языка XSLT (Extensible Stylesheet Language Transformations — расширяемый язык преобразования XML-документов). Более подробно язык XSLT рассматривается в последнем разделе этой главы, а до тех пор допустим, что приведенный ниже код каким-то волшебным образом позволяет получить данные, выводимые в формате XML.

Над документом выполняется холостое преобразование, а результат записывается в поток вывода. Чтобы включить узел `DOCTYPE` в выводимые данные, следует также указать идентификаторы `SYSTEM` и `PUBLIC` в качестве свойств вывода.

```
// построить объект холостого преобразования
Transformer t = TransformerFactory.newInstance().newTransformer();
// установить свойства вывода, чтобы получить узел DOCTYPE
t.setOutputProperty(OutputKeys.DOCUMENT_TYPE_NAME, "public");
t.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "yes");
// установить отступ
t.setOutputProperty(OutputKeys.INDENT, "yes");
t.setOutputProperty(OutputKeys.METHOD, "xml");
t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
// выполнить холостое преобразование и вывести результат в файл
t.transform(new DOMSource(doc),
    new StreamResult(new FileOutputStream(file)));}

```

Еще один способ записи XML-документов состоит в применении интерфейса `LSSerializer`. Для получения экземпляра класса, реализующего этот интерфейс, служит следующий фрагмент кода:

```
DOMImplementation impl = doc.getImplementation();
DOMImplementationLS implLS = (DOMImplementationLS)
    impl.getFeature("LS", "3.0");
LSSerializer ser = implLS.createLSSerializer();
```

Если требуется ввести пробелы и разрывы строк в документ, для этого достаточно установить следующий флаг:

```
ser.getDomConfig().setParameter("format-pretty-print", true);
```

И тогда преобразовать документ в символьную строку не составит особого труда:

```
String str = ser.writeToString(doc);
```

Если же требуется вывести документ непосредственно в файл, для этого нужно создать объект типа `LSOutput` следующим образом:

```
LSOutput out = implLS.createLSOutput();
out.setEncoding("UTF-8");
out.setByteStream(Files.newOutputStream(path));
ser.write(doc, out);
```

3.7.4. Пример формирования файла в формате SVG

В листинге 3.10, приведенном в следующем разделе, представлен исходный код примера программы для вывода XML-документа. Эта программа рисует картину в модернистском стиле из произвольного набора прямоугольников разного цвета (рис. 3.6). Для сохранения результатов используется формат SVG (Scalable Vector Graphics — масштабируемая векторная графика). По существу, формат SVG является разновидностью формата XML и служит для описания сложной графики в машинно-независимом виде. Дополнительные сведения об этом формате можно найти по адресу <http://www.w3c.org/Graphics/SVG>. Для просмотра файлов в формате SVG достаточно воспользоваться любым современным браузером.

Мы не будем вдаваться в подробности формата SVG, отсылая интересующихся за дополнительными сведениями по указанному выше адресу. Для целей рассматриваемого здесь примера достаточно знать, каким образом набор цветных прямоугольников размечается в формате SVG. Ниже приведен пример такой разметки.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20000802//EN"
```

```

"http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd">
<svg xmlns="http://www.w3.org/2000/svg" width="300" height="150">
<rect x="231" y="61" width="9" height="12" fill="#6e4a13"/>
<rect x="107" y="106" width="56" height="5" fill="#c406be"/>
. . .
</svg>

```

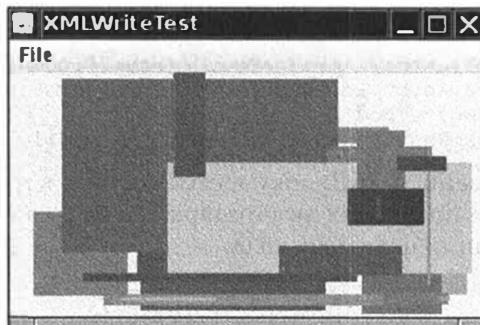


Рис. 3.6. Изображение в модернистском стиле, сохраняемое в формате SVG

Как видите, каждый прямоугольник описывается в виде узла `rect`, атрибуты которого задают координаты, ширину, высоту и цвет прямоугольника. Цвет заливки прямоугольников обозначается в виде значений основных цветов RGB в шестнадцатеричной форме.

На заметку! В формате SVG широко применяются атрибуты. На самом деле некоторые из них имеют очень сложную структуру. В качестве примера ниже приведен элемент разметки контура,

`<path d="M 100 100 L 300 100 L 200 300 z">`

где `M` обозначает команду `moveto` (перейти), `L` — команду `lineto` (нарисовать линию), `z` — команду `closepath` (замкнуть контур). Вероятно, создатели формата SVG не особенно доверяли формату XML. Очевидно, что вместо таких сложных атрибутов следовало бы использовать элементы разметки в коде XML.

javax.xml.parsers.DocumentBuilder 1.4

- `Document newDocument()`
Возвращает пустой документ.

org.w3c.dom.Document 1.4

- `Element createElement(String name)`
- `Element createElementNS(String uri, String qname)`
Создают элемент с заданным именем.
- `Text createTextNode(String data)`
Создает текстовый узел с указанными данными.

org.w3c.dom.Node 1.4

- **Node appendChild(Node child)**

Присоединяет узел к списку его дочерних узлов. Возвращает присоединенный узел.

org.w3c.dom.Element 1.4

- **void setAttribute(String name, String value)**
- **void setAttributeNS(String uri, String qname, String value)**

Устанавливают заданное значение в атрибуте с указанным именем.

Параметры:	uri	URI пространства имен или пустое значение null
	qname	Уточненное имя. Если оно имеет префикс, то параметр uri не должен принимать пустое значение null
	value	Значение атрибута

javax.xml.transform.TransformerFactory 1.4

- **static TransformerFactory newInstance()**

Возвращает экземпляр класса **TransformerFactory**.

- **Transformer newTransformer()**

Возвращает экземпляр класса **Transformer**, выполняющий тождественное (холостое) преобразование, не предполагающее никаких действий.

javax.xml.transform.Transformer 1.4

- **void setOutputProperty(String name, String value)**

Задает свойство вывода. Перечень этих свойств можно найти по адресу <http://www.w3c.org/TR/xslt#output>. А ниже перечислены наиболее употребительные свойства вывода.

doctype-public	Идентификатор PUBLIC , используемый в объявлении DOCTYPE
-----------------------	--

doctype-system	Идентификатор SYSTEM , используемый в объявлении DOCTYPE
-----------------------	--

indent	Принимает значение "yes" или "no"
---------------	-----------------------------------

method	Принимает значение "xml", "html", "text" или специальное строковое значение
---------------	---

- **void transform(Source from, Result to)**

Выполняет преобразование XML-документа.

javax.xml.transform.dom.DOMSource 1.4

- **DOMSource(Node n)**

Создает источник данных из заданного узла. Обычно параметр *n* обозначает узел документа.

javax.xml.transform.stream.StreamResult 1.4

- **StreamResult(File f)**
- **StreamResult(OutputStream out)**
- **StreamResult(Writer out)**
- **StreamResult(String systemID)**

Создают поток вывода результатов преобразования на основе указанного файла, потока вывода, потока записи или системного идентификатора (как правило, это относительный или абсолютный URL).

3.7.5. Запись XML-документов средствами StAX

В предыдущем разделе было показано, как XML-документ формируется путем записи дерева модели DOM. Но если дерево модели DOM нигде больше не используется, то такой способ оказывается не особенно эффективным. Прикладной программный интерфейс StAX API позволяет записывать дерево формируемого документа непосредственно в формате XML. Для этого следует создать поток записи типа `XMLStreamWriter` из потока вывода типа `OutputStream`, как показано ниже.

```
XMLOutputFactory factory = XMLOutputFactory.newInstance();
XMLStreamWriter writer = factory.createXMLStreamWriter(out);
```

А для того чтобы создать и вывести заголовок XML-документа, нужно вызвать сначала метод

```
writer.writeStartDocument()
```

а затем метод

```
writer.writeStartElement(name);
```

Далее, для вывода атрибутов следует вызвать приведенный ниже метод.

```
writer.writeAttribute(name, value);
```

Теперь можно вывести дочерние элементы разметки, снова вызвав метод `writeStartElement()`, или записать символы, вызвав следующий метод:

```
writer.writeCharacters(text);
```

После записи всех дочерних узлов следует вызвать приведенный ниже метод, который закроет текущий элемент разметки.

```
writer.writeEndElement();
```

Чтобы записать элемент разметки без дочерних элементов (например, элемент ``), следует вызвать следующий метод:

```
writer.writeEmptyElement(name);
```

И наконец, для завершения записи в конце документа вызывается приведенный ниже метод. Этот метод закрывает все открытые элементы разметки.

```
writer.writeEndDocument();
```

Поток записи типа `XMLStreamWriter` придется все же закрыть вручную. Ведь интерфейс `XMLStreamWriter` не расширяет интерфейс `AutoCloseable`.

Как и при подходе с применением модели DOM и языка XSLT, в данном случае можно не особенно беспокоиться о пропуске символов в значениях атрибутов и символьных данных. Но в этом случае существует вероятность того, что XML-документ будет сформирован не совсем удачно, например, со многими корневыми узлами. Кроме того, в текущей версии прикладного программного интерфейса StAX API не поддерживается вывод XML-документов с отступами.

В примере программы из листинга 3.10 демонстрируется применение каждого из рассмотренных выше способов записи XML-документов. А в листингах 3.11 и 3.12 приведен исходный код классов фрейма и его компонентов для рисования прямоугольников, заполняемых цветом.

Листинг 3.10. Исходный код из файла write/XMLWriteTest.java

```
1 package write;
2
3 import java.awt.*;
4 import javax.swing.*;
5
6 /**
7  * В этой программе демонстрируется запись XML-документа в файл.
8  * Сохраняемый файл описывает модернистский рисунок в формате SVG
9  * @version 1.12 2016-04-27
10 * @author Cay Horstmann
11 */
12 public class XMLWriteTest
13 {
14     public static void main(String[] args)
15     {
16         EventQueue.invokeLater(() ->
17         {
18             JFrame frame = new XMLWriteFrame();
19             frame.setTitle("XMLWriteTest");
20             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21             frame.setVisible(true);
22         });
23     }
24 }
```

Листинг 3.11. Исходный код из файла write/XMLWriteFrame.java

```
1 package write;
2
3 import java.io.*;
4 import java.nio.file.*;
5
6 import javax.swing.*;
7 import javax.xml.stream.*;
8 import javax.xml.transform.*;
9 import javax.xml.transform.dom.*;
```

```
10 import javax.xml.transform.stream.*;
11
12 import org.w3c.dom.*;
13
14 /**
15 * Фрейм с компонентом для отображения модернистского рисунка
16 */
17 public class XMLWriteFrame extends JFrame
18 {
19     private RectangleComponent comp;
20     private JFileChooser chooser;
21
22     public XMLWriteFrame()
23     {
24         chooser = new JFileChooser();
25
26         // ввести компонент в фрейм
27
28         comp = new RectangleComponent();
29         add(comp);
30
31         // установить строку меню
32
33         JMenuBar menuBar = new JMenuBar();
34         setJMenuBar(menuBar);
35
36         JMenu menu = new JMenu("File");
37         menuBar.add(menu);
38
39         JMenuItem newItem = new JMenuItem("New");
40         menu.add(newItem);
41         newItem.addActionListener(event -> comp.newDrawing());
42
43         JMenuItem saveItem = new JMenuItem("Save with DOM/XSLT");
44         menu.add(saveItem);
45         saveItem.addActionListener(event -> saveDocument());
46
47         JMenuItem saveStAXItem = new JMenuItem("Save with StAX");
48         menu.add(saveStAXItem);
49         saveStAXItem.addActionListener(event -> saveStAX());
50
51         JMenuItem exitItem = new JMenuItem("Exit");
52         menu.add(exitItem);
53         exitItem.addActionListener(event -> System.exit(0));
54         pack();
55     }
56
57 /**
58 * Сохраняет рисунок в формате SVG средствами DOM/XSLT
59 */
60 public void saveDocument()
61 {
62     try
63     {
64         if (chooser.showSaveDialog(this)
65             != JFileChooser.APPROVE_OPTION) return;
66         File file = chooser.getSelectedFile();
67         Document doc = comp.buildDocument();
68         Transformer t = TransformerFactory.newInstance()
69             .newTransformer();
```

```

70     t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM,
71         "http://www.w3.org/TR/2000/CR-SVG-20000802/
72             DTD/svg-20000802.dtd");
73     t.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC,
74         "-//W3C//DTD SVG 20000802//EN");
75     t.setOutputProperty(OutputKeys.INDENT, "yes");
76     t.setOutputProperty(OutputKeys.METHOD, "xml");
77     t.setOutputProperty(
78         "{http://xml.apache.org/xslt}indent-amount", "2");
79     t.transform(new DOMSource(doc), new
80             StreamResult(Files.newOutputStream(file.toPath())));
81 }
82 catch (TransformerException | IOException ex)
83 {
84     ex.printStackTrace();
85 }
86 }
87 /**
88 * Сохраняет рисунок в формате SVG средствами StAX
89 */
90 public void saveStAX()
91 {
92     if (chooser.showSaveDialog(this)
93         != JFileChooser.APPROVE_OPTION) return;
94     File file = chooser.getSelectedFile();
95     XMLErrorFactory factory = XMLErrorFactory.newInstance();
96     try
97     {
98         XMLStreamWriter writer = factory.createXMLStreamWriter(
99             Files.newOutputStream(file.toPath()));
100        try
101        {
102            comp.writeDocument(writer);
103        }
104        finally
105        {
106            writer.close(); // не закрывается автоматически
107        }
108    }
109    catch (XMLStreamException | IOException ex)
110    {
111        ex.printStackTrace();
112    }
113 }
114 }
115 }
116 }
```

Листинг 3.12. Исходный код из файла write/RectangleComponent.java

```

1 package write;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import java.util.*;
6 import javax.swing.*;
7 import javax.xml.parsers.*;
8 import javax.xml.stream.*;
9 import org.w3c.dom.*;
```

```
11 /**
12  * Компонент, отображающий ряд окрашенных прямоугольников
13 */
14 public class RectangleComponent extends JComponent
15 {
16     private static final Dimension
17         PREFERRED_SIZE = new Dimension(300, 200);
18
19     private java.util.List<Rectangle2D> rects;
20     private java.util.List<Color> colors;
21     private Random generator;
22     private DocumentBuilder builder;
23
24     public RectangleComponent()
25     {
26         rects = new ArrayList<>();
27         colors = new ArrayList<>();
28         generator = new Random();
29
30         DocumentBuilderFactory factory =
31             DocumentBuilderFactory.newInstance();
32         factory.setNamespaceAware(true);
33         try
34         {
35             builder = factory.newDocumentBuilder();
36         }
37         catch (ParserConfigurationException e)
38         {
39             e.printStackTrace();
40         }
41     }
42
43     /**
44      * Создать новый произвольный рисунок прямоугольника
45     */
46     public void newDrawing()
47     {
48         int n = 10 + generator.nextInt(20);
49         rects.clear();
50         colors.clear();
51         for (int i = 1; i <= n; i++)
52         {
53             int x = generator.nextInt(getWidth());
54             int y = generator.nextInt(getHeight());
55             int width = generator.nextInt(getWidth() - x);
56             int height = generator.nextInt(getHeight() - y);
57             rects.add(new Rectangle(x, y, width, height));
58             int r = generator.nextInt(256);
59             int g = generator.nextInt(256);
60             int b = generator.nextInt(256);
61             colors.add(new Color(r, g, b));
62         }
63         repaint();
64     }
65
66     public void paintComponent(Graphics g)
67     {
68         if (rects.size() == 0) newDrawing();
69         Graphics2D g2 = (Graphics2D) g;
```

```
70     // нарисовать все прямоугольники
71     for (int i = 0; i < rects.size(); i++)
72     {
73         g2.setPaint(colors.get(i));
74         g2.fill(rects.get(i));
75     }
76 }
77
78 /**
79 * Создает документ формата SVG с текущим рисунком
80 * @return Возвращает дерево DOM для документа формата SVG
81 */
82 public Document buildDocument()
83 {
84     String namespace = "http://www.w3.org/2000/svg";
85     Document doc = builder.newDocument();
86     Element svgElement = doc.createElementNS(namespace, "svg");
87     doc.appendChild(svgElement);
88     svgElement.setAttribute("width", "" + getWidth());
89     svgElement.setAttribute("height", "" + getHeight());
90     for (int i = 0; i < rects.size(); i++)
91     {
92         Color c = colors.get(i);
93         Rectangle2D r = rects.get(i);
94         Element rectElement = doc.createElementNS(
95             namespace, "rect");
96         rectElement.setAttribute("x", "" + r.getX());
97         rectElement.setAttribute("y", "" + r.getY());
98         rectElement.setAttribute("width", "" + r.getWidth());
99         rectElement.setAttribute("height", "" + r.getHeight());
100        rectElement.setAttribute("fill", String.format("#%06x",
101                                         c.getRGB() & 0xFFFFFFFF));
102        svgElement.appendChild(rectElement);
103    }
104    return doc;
105 }
106
107 /**
108 * Записывает документ формата SVG с текущим рисунком
109 * @param writer Место назначения документа
110 */
111 public void writeDocument(XMLStreamWriter writer)
112     throws XMLStreamException
113 {
114     writer.writeStartDocument();
115     writer.writeDTD(
116         "<!DOCTYPE svg PUBLIC \"-//W3C//DTD SVG 20000802//EN\" "
117         + "\\" + "http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/
118             svg-20000802.dtd\\>\"");
119     writer.writeStartElement("svg");
120     writer.writeDefaultNamespace("http://www.w3.org/2000/svg");
121     writer.writeAttribute("width", "" + getWidth());
122     writer.writeAttribute("height", "" + getHeight());
123     for (int i = 0; i < rects.size(); i++)
124     {
125         Color c = colors.get(i);
126         Rectangle2D r = rects.get(i);
127         writer.writeEmptyElement("rect");
128         writer.writeAttribute("x", "" + r.getX());
129         writer.writeAttribute("y", "" + r.getY());
```

```

130     writer.writeAttribute("width", "" + r.getWidth());
131     writer.writeAttribute("height", "" + r.getHeight());
132     writer.writeAttribute("fill", String.format("#%06x",
133                                         c.getRGB() & 0xFFFFFF));
134 }
135     writer.writeEndDocument(); // closes svg element
136 }
137
138 public Dimension getPreferredSize() { return PREFERRED_SIZE; }
139 }
```

javax.xml.stream.XMLOutputFactory 6

- **static XMLOutputFactory newInstance()**
Возвращает экземпляр класса `XMLOutputFactory`.
- **XMLStreamWriter createXMLStreamWriter(OutputStream in)**
- **XMLStreamWriter createXMLStreamWriter(OutputStream in, String characterEncoding)**
- **XMLStreamWriter createXMLStreamWriter(Writer in)**
- **XMLStreamWriter createXMLStreamWriter(Result in)**
Создают поток, записывающий данные в указанный поток вывода, поток записи или результат типа JAXP.

javax.xml.stream.XMLStreamWriter 6

- **void writeStartDocument()**
- **void writeStartDocument(String xmlVersion)**
- **void writeStartDocument(String encoding, String xmlVersion)**
Записывают инструкцию обработки в начале XML-документа. Следует, однако, иметь в виду, что параметр `encoding` указывается только для записи атрибута и не задает кодировку символов в выводимых данных.
- **void setDefaultNamespace(String namespaceURI)**
- **void setPrefix(String prefix, String namespaceURI)**
Задают пространство имен по умолчанию или пространство имен, связанное с префиксом. Объявление действительно для текущего элемента или для корня документа, если ни один элемент не записан.
- **void writeStartElement(String localName)**
- **void writeStartElement(String namespaceURI, String localName)**
Записывают первоначальный дескриптор, заменяя параметр `namespaceURI` соответствующим префиксом.
- **void writeEndElement()**
Закрывает текущий элемент разметки.
- **void writeEndDocument()**
Закрывает все открытые элементы разметки.
- **void writeEmptyElement(String localName)**
- **void writeEmptyElement(String namespaceURI, String localName)**
Записывают самозакрывающийся дескриптор, заменяя параметр `namespaceURI` соответствующим префиксом.

javax.xml.stream.XMLStreamWriter 6 (окончание)

- **void writeAttribute(String localName, String value)**
- **void writeAttribute(String namespaceURI, String localName, String value)**
Записывают атрибут для текущего элемента разметки, заменяя параметр `namespaceURI` соответствующим префиксом.
- **void writeCharacters(String text)**
Записывает символьные данные.
- **void writeCData(String text)**
Записывает раздел `CDATA`.
- **void writeDTD(String dtd)**
Записывает символьную строку `dtd`, которая должна содержать объявление `DOCTYPE`.
- **void writeComment(String comment)**
Записывает комментарий.
- **void close()**
Закрывает поток записи.

3.8. Преобразование XML-документов языковыми средствами XSLT

Язык преобразования XML-документов (XSLT) позволяет определять правила преобразования подобных документов в другие форматы, включая простой текст, XHTML или любую другую разновидность формата XML. Язык XSLT обычно применяется для перевода из одной машиночитаемой разновидности формата XML в другую машиночитаемую или удобочитаемую (для человека) разновидность этого формата.

Для этой цели следует создать таблицу стилей XSLT, описывающую преобразование XML-документов в какой-нибудь другой формат. Процессор XSLT сначала читает XML-документ и таблицу стилей, а затем выдает желаемый результат (рис. 3.7).

Спецификация языка XSLT довольно сложная, и ее описанию посвящены целые книги. Здесь недостаточно места для описания всех языковых средств XSLT, поэтому мы рассмотрим лишь наглядный пример их применения. Подробнее ознакомиться с особенностями языка XSLT можно в книге *Essential XML* Дона Бокса и др., упоминавшейся в начале этой главы, а спецификация языка XSLT доступна по адресу www.w3.org/TR/xslt.

Допустим, требуется преобразовать XML-документ с записями о сотрудниках в HTML-документ. Ниже приведена разметка исходного XML-документа.

```
<staff>
  <employee>
    <name>Carl Cracker</name>
    <salary>75000</salary>
    <hiredate year="1987" month="12" day="15"/>
  </employee>
  <employee>
    <name>Harry Hacker</name>
    <salary>50000</salary>
    <hiredate year="1989" month="10" day="1"/>
```

```

</employee>
<employee>
  <name>Tony Tester</name>
  <salary>40000</salary>
  <hiredate year="1990" month="3" day="15"/>
</employee>
</staff>

```

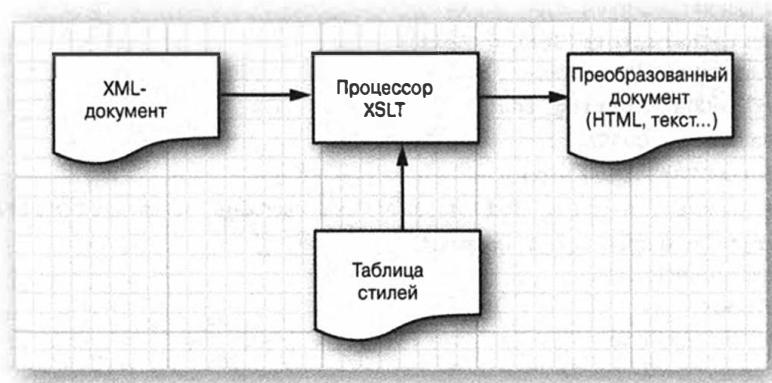


Рис. 3.7. Преобразование XML-документа языковыми средствами XSLT

Из этой разметки желательно получить следующую HTML-таблицу:

Carl Cracker	\$75000.0	1987-12-15
Harry Hacker	\$50000.0	1989-10-1
Tony Tester	\$40000.0	1990-3-15

Вот как выглядит таблица стилей с шаблонами преобразования:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:output method="html"/>
  шаблон1
  шаблон2
  . . .
</xsl:stylesheet>

```

В данном примере элемент разметки `xsl:output` содержит атрибут `method` со значением "html" для преобразования в формат HTML. Для данного атрибута можно также задать значения `xml` и `text`. Типичный шаблон имеет следующий вид:

```

<xsl:template match="/staff/employee">
  <tr><xsl:apply-templates/></tr>
</xsl:template>

```

Значением атрибута `match` является выражение XPath. Данный шаблон означает, что при обнаружении узла во множестве `/staff/employee`, указанном в выражении XPath, нужно выполнить следующие действия.

1. Сформировать символьную строку с открывающим дескриптором `<tr>`.
2. Применить шаблоны при обработке дочерних узлов данного узла.
3. После обработки всех дочерних узлов сформировать символьную строку с закрывающим дескриптором `</tr>`.

Иными словами, этот шаблон заключает каждую запись о сотрудниках в дескрипторы строк HTML-таблицы.

Процессор XSLT начинает обработку XML-документа с проверки корневого элемента разметки. Если узел совпадает с одним из шаблонов, соответствующий шаблон сразу же применяется. (При совпадении с несколькими шаблонами используется шаблон с наибольшей степенью соответствия. Дополнительные сведения по данному вопросу приведены в спецификации языка XSLT по адресу <http://www.w3.org/TR/xslt>.) Если совпадение с шаблоном не обнаружено, процессор XSLT выполняет действие, задаваемое по умолчанию. Так, содержимое текстовых узлов по умолчанию включается в выводимый результат, а для элементов разметки выводимый результат не формируется, но продолжается обработка дочерних элементов.

В качестве примера ниже приведен шаблон преобразования узлов `name` из XML-документа с данными о сотрудниках.

```
<xsl:template match="/staff/employee/name">
  <td><xsl:apply-templates/></td>
</xsl:template>
```

Как видите, шаблон формирует дескрипторы `<td>...</td>` и предписывает процессору XSLT рекурсивно обойти дочерние узлы элемента разметки `name`. У этого элемента имеется только один дочерний текстовый узел. Когда процессор обходит узел, он возвращает его текстовое содержимое, если, конечно, отсутствуют другие совпавшие шаблоны.

Для копирования значений атрибутов в выводимый результат придется задать более сложный шаблон:

```
<xsl:template match="/staff/employee/hiredate">
  <td><xsl:value-of select="@year"/>-<xsl:value-of
    select="@month"/>-<xsl:value-of select="@day"/></td>
</xsl:template>
```

При обработке узла `hiredate` по этому шаблону будут сформированы перечисленные ниже элементы разметки таблицы.

1. Дескриптор `<td>`.
2. Значение атрибута `year`.
3. Дефис.
4. Значение атрибута `month`.
5. Дефис.
6. Значение атрибута `day`.
7. Дескриптор `</td>`.

Оператор `xsl:value-of` вычисляет строковое значение множества узлов, указанного с помощью значения XPath атрибута `select`. В этом случае путь определяется относительно текущего узла. Множество узлов преобразуется в символьную строку путем сцепления строковых значений из всех узлов. Строковым значением атрибута является его значение, строковым значением текстового узла — его содержимое, а строковым значением элемента разметки — сцепление строковых значений его дочерних узлов (но не атрибутов).

В листинге 3.13 приведена таблица стилей для преобразования XML-документа с записями о сотрудниках в HTML-таблицу.

Листинг 3.13. Исходный код из файла transform/makehtml.xsl

```

1  <?xml version="1.0" encoding="ISO-8859-1"?>
2
3  <xsl:stylesheet
4      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
5      version="1.0">
6
7      <xsl:output method="html"/>
8
9      <xsl:template match="/staff">
10         <table border="1"><xsl:apply-templates/></table>
11     </xsl:template>
12
13     <xsl:template match="/staff/employee">
14         <tr><xsl:apply-templates/></tr>
15     </xsl:template>
16
17     <xsl:template match="/staff/employee/name">
18         <td><xsl:apply-templates/></td>
19     </xsl:template>
20
21     <xsl:template match="/staff/employee/salary">
22         <td>$<xsl:apply-templates/></td>
23     </xsl:template>
24
25     <xsl:template match="/staff/employee/hiredate">
26         <td><xsl:value-of select="@year"/>-<xsl:value-of
27             select="@month"/>-<xsl:value-of select="@day"/></td>
28     </xsl:template>
29
30 </xsl:stylesheet>
```

А в листинге 3.14 приведены шаблоны для различных преобразований того же самого XML-документа в обычный текст в знакомом уже формате файла свойств.

```

employee.1.name=Carl Cracker
employee.1.salary=75000.0
employee.1.hiredate=1987-12-15
employee.2.name=Harry Hacker
employee.2.salary=50000.0
employee.2.hiredate=1989-10-1
employee.3.name=Tony Tester
employee.3.salary=40000.0
employee.3.hiredate=1990-3-15
```

Листинг 3.14. Исходный код из файла transform/makeprop.xsl

```

1  <?xml version="1.0"?>
2
3  <xsl:stylesheet
4      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
5      version="1.0">
6
7      <xsl:output method="text" omit-xml-declaration="yes"/>
8
9      <xsl:template match="/staff/employee">
10 employee.<xsl:value-of select="position()" />
11 >.<xsl:value-of select="name/text()"/>
12 employee.<xsl:value-of select="position()" />
13 >.<xsl:value-of select="salary/text()"/>
14 employee.<xsl:value-of select="position()" />
15 >.<xsl:value-of select="hiredate/@year" />-
16 >.<xsl:value-of select="hiredate/@month" />-
17 >.<xsl:value-of select="hiredate/@day" />
18     </xsl:template>
19
20 </xsl:stylesheet>

```

В данном примере используется функция `position()`, которая выдает расположение текущего узла относительно родительского. Чтобы получить выводимый результат в совершенно другом виде, достаточно внести соответствующие изменения в таблицу стилей. Таким образом, XML-документ можно благополучно применять для представления данных, не особенно заботясь о том, в каком именно формате они требуются в прикладной программе. Для формирования данных в нужном формате достаточно воспользоваться средствами XSLT.

Преобразования средствами XSLT совсем не трудно организовать на платформе Java. С этой целью нужно создать сначала отдельную фабрику преобразователей для каждой таблицы стилей, а затем получить объект типа `Transformer` и передать ему преобразуемые данные:

```

File styleSheet = new File(filename);
StreamSource styleSource = new StreamSource(styleSheet);
Transformer t = TransformerFactory.newInstance().
newTransformer(styleSource);
t.transform(source, result);

```

Параметры, передаваемые методу `transform()`, представляют собой экземпляры классов, реализующих интерфейсы `Source` и `Result`. Так, у интерфейса `Source` имеются реализации источника данных в следующих классах:

```

DOMSource
SAXSource
StAXSource
StreamSource

```

Потоковый источник данных типа `StreamSource` можно создать из файла, потока ввода, потока чтения или URL, а источник данных типа `DOMSource` — из узла дерева модели DOM. Так, в примере программы из листинга 3.11 выполнялось следующее тождественное преобразование:

```
t.transform(new DOMSource(doc), result);
```

А в рассматриваемом здесь примере программы применяется другой, более интересный подход. Вместо уже существующего XML-файла создается поток, читающий данные в формате XML, имитируя их SAX-анализ с инициированием соответствующих SAX-событий. По существу, поток чтения данных формата XML выполняет чтение из однородного входного файла со следующим содержимым:

```
Carl Cracker|75000.0|1987|12|15
Harry Hacker|50000.0|1989|10|1
Tony Tester|40000.0|1990|3|15
```

По мере обработки вводимых данных формата XML в потоке чтения инициируются SAX-события. Ниже представлен фрагмент исходного кода метода `parse()` из класса `EmployeeReader`, реализующего интерфейс `XMLReader`.

```
AttributesImpl attributes = new AttributesImpl();
handler.startDocument();
handler.startElement("", "staff", "staff", attributes);
while ((line = in.readLine()) != null)
{
    handler.startElement("", "employee", "employee", attributes);
    StringTokenizer t = new StringTokenizer(line, "|");
    handler.startElement("", "name", "name", attributes);
    String s = t.nextToken();
    handler.characters(s.toCharArray(), 0, s.length());
    handler.endElement("", "name", "name");
    ...
    handler.endElement("", "employee", "employee");
}
handler.endElement("", rootElement, rootElement);
handler.endDocument();
```

Источник типа `SAXSource` для преобразования данных создается из потока чтения XML-данных следующим образом:

```
t.transform(new SAXSource(new EmployeeReader(),
    new InputSource(new FileInputStream(filename))), result);
```

Такой искусственный прием как нельзя лучше подходит для преобразования унаследованных данных в формат XML. Безусловно, для большинства приложений XSLT вводимые данные уже находятся в формате XML. А это позволяет просто вызвать метод `transform()` для объекта типа `SAXSource`:

```
t.transform(new StreamSource(file), result);
```

Результатом такого преобразования оказывается объект одного из следующих трех классов, реализующих интерфейс `Result` в библиотеке Java:

```
DOMResult
SAXResult
StreamResult
```

Для сохранения результата в виде древовидной структуры DOM используется объект типа `DocumentBuilder`. С его помощью генерируется новый узел документа, заключаемый в оболочку типа `DOMResult`:

```
Document doc = builder.newDocument();
t.transform(source, new DOMResult(doc));
```

И наконец, для вывода полученного результата в файл используется объект типа `StreamResult`:

```
t.transform(source, new StreamResult(file));
```

В листинге 3.15 приведен весь исходный код рассмотренного здесь примера программы.

Листинг 3.15. Исходный код из файла transform/TransformTest.java

```
1 package transform;
2
3 import java.io.*;
4 import java.nio.file.*;
5 import java.util.*;
6 import javax.xml.transform.*;
7 import javax.xml.transform.sax.*;
8 import javax.xml.transform.stream.*;
9 import org.xml.sax.*;
10 import org.xml.sax.helpers.*;
11
12 /**
13 * В этой программе демонстрируются преобразования языковыми
14 * средствами XSLT. Преобразования выполняются над записями о
15 * сотрудниках из файла employee.dat в формат XML. Для этого
16 * нужно указать таблицу стилей в командной строке, например,
17 * следующим образом:
18 * java TransformTest transform/makeprop.xsl
19 * @version 1.03 2016-04-27
20 * @author Cay Horstmann
21 */
22 public class TransformTest
23 {
24     public static void main(String[] args) throws Exception
25     {
26         Path path;
27         if (args.length > 0) path = Paths.get(args[0]);
28         else path = Paths.get("transform", "makehtml.xsl");
29         try (InputStream styleIn = Files.newInputStream(path))
30         {
31             StreamSource styleSource = new StreamSource(styleIn);
32
33             Transformer t = TransformerFactory.newInstance()
34                     .newTransformer(styleSource);
35             t.setOutputProperty(OutputKeys.INDENT, "yes");
36             t.setOutputProperty(OutputKeys.METHOD, "xml");
37             t.setOutputProperty(
38                 "http://xml.apache.org/xslt-indent-amount", "2");
39
40             try (InputStream docIn = Files.newInputStream(Paths.get(
41                         "transform", "employee.dat")))
42             {
43                 t.transform(new SAXSource(new EmployeeReader(),
44                                         new InputSource(docIn)),
45                                         new StreamResult(System.out));
46             }
47         }
48     }
49
50 /**
51 * Этот класс читает однородный файл employee.dat и уведомляет
52 * о событиях в SAX-анализаторе, как будто он сам выполнил
53 * синтаксический анализ XML-документа
```

```
54  */
55 class EmployeeReader implements XMLReader
56 {
57     private ContentHandler handler;
58
59     public void parse(InputSource source)
60             throws IOException, SAXException
61     {
62         InputStream stream = source.getByteStream();
63         BufferedReader in = new BufferedReader(
64             new InputStreamReader(stream));
65         String rootElement = "staff";
66         AttributesImpl atts = new AttributesImpl();
67
68         if (handler == null)
69             throw new SAXException("No content handler");
70
71         handler.startDocument();
72         handler.startElement("", rootElement, rootElement, atts);
73         String line;
74         while ((line = in.readLine()) != null)
75     {
76             handler.startElement("", "employee", "employee", atts);
77             StringTokenizer t = new StringTokenizer(line, "|");
78
79             handler.startElement("", "name", "name", atts);
80             String s = t.nextToken();
81             handler.characters(s.toCharArray(), 0, s.length());
82             handler.endElement("", "name", "name");
83
84             handler.startElement("", "salary", "salary", atts);
85             s = t.nextToken();
86             handler.characters(s.toCharArray(), 0, s.length());
87             handler.endElement("", "salary", "salary");
88
89             atts.addAttribute("", "year", "year", "CDATA",
90                             t.nextToken());
91             atts.addAttribute("", "month", "month", "CDATA",
92                             t.nextToken());
93             atts.addAttribute("", "day", "day", "CDATA",
94                             t.nextToken());
95             handler.startElement("", "hiredate", "hiredate", atts);
96             handler.endElement("", "hiredate", "hiredate");
97             atts.clear();
98
99             handler.endElement("", "employee", "employee");
100 }
101
102         handler.endElement("", rootElement, rootElement);
103         handler.endDocument();
104     }
105
106     public void setContentHandler(ContentHandler newValue)
107     {
108         handler = newValue;
109     }
110
111     public ContentHandler getContentHandler()
112     {
113         return handler;
114     }
```

```
115
116 // следующие методы являются всего лишь холостыми реализациями
117 public void parse(String systemId) throws IOException,
118     SAXException {}
119 public void setErrorHandler(ErrorHandler handler) {}
120 public ErrorHandler getErrorHandler() { return null; }
121 public void setDTDHandler(DTDHandler handler) {}
122 public DTDHandler getDTDHandler() { return null; }
123 public void setEntityResolver(EntityResolver resolver) {}
124 public EntityResolver getEntityResolver() { return null; }
125 public void setProperty(String name, Object value) {}
126 public Object getProperty(String name) { return null; }
127 public void setFeature(String name, boolean value) {}
128 public boolean getFeature(String name) { return false; }
129 }
```

javax.xml.transform.TransformerFactory 1.4

- **Transformer newTransformer(Source styleSheet)**

Возвращает экземпляр класса **Transformer**,читывающий таблицу стилей из указанного источника.

javax.xml.transform.stream.StreamSource 1.4

- **StreamSource(File f)**
- **StreamSource(InputStream in)**
- **StreamSource(Reader in)**
- **StreamSource(String systemID)**

Создают потоковый источник данных из указанного файла, потока ввода, потока чтения или системного идентификатора (обычно это относительный или абсолютный URL).

javax.xml.transform.sax.SAXSource 1.4

- **SAXSource(XMLReader reader, InputSource source)**

Создает SAX-источник, получающий вводимые данные из указанного источника, используя заданный поток чтения для синтаксического анализа данных.

org.xml.sax.XMLReader 1.4

- **void setContentHandler(ContentHandler handler)**

Устанавливает обработчик, который уведомляется о событиях, наступающих при синтаксическом анализе вводимых данных.

- **void parse(InputSource source)**

Анализирует данные, вводимые из указанного источника, и передает обработчику содержащего события, наступающие при синтаксическом анализе этих данных.

javax.xml.transform.dom.DOMResult 1.4

- DOMResult(Node n)**

Создает источник данных из заданного узла. Обычно в качестве параметра *n* указывается узел документа.

org.xml.sax.helpers.AttributesImpl 1.4

- void addAttribute(String uri, String lname, String qname, String type, String value)**

Вводит атрибут в коллекцию атрибутов.

Параметры:	<i>uri</i>	URI пространства имен
	<i>lname</i>	Локальное имя без префикса
	<i>qname</i>	Уточненное имя с префиксом
	<i>type</i>	Один из следующих типов: "CDATA", "ID", "IDREF", "IDREFS", "NMTOKEN", "NMTOKENS", "ENTITY", "ENTITIES" или "NOTATION"
	<i>value</i>	Значение атрибута

- void clear()**

Удаляет все атрибуты из данной коллекции.

Этим примером завершается обсуждение особенностей поддержки XML в библиотеке Java. Теперь у вас должно сложиться ясное представление о возможностях XML, включая автоматизированный синтаксический анализ и проверку достоверности, а также эффективный механизм преобразования XML-документов. Естественно, что всю эту технологию вам удастся поставить себе на службу, если вы тщательно разработаете свои форматы XML. Для этого вы должны обеспечить способность ваших форматов XML удовлетворять всем насущным производственным потребностям, их устойчивость с течением времени, а также выяснить готовность ваших деловых партнеров принимать от вас XML-документы. Решение всех этих вопросов может оказаться гораздо сложнее, чем умелое обращение с синтаксическими анализаторами, определениями DTD или преобразованиями, выполняемыми средствами XSLT.

В следующей главе будут обсуждаться вопросы сетевого программирования на платформе Java. Сначала мы рассмотрим основные положения о сетевых сокетах, а затем перейдем к высокоуровневым протоколам для электронной почты и Всемирной паутины.

4

ГЛАВА

Работа в сети

В этой главе...

- ▶ Подключение к серверу
- ▶ Реализация серверов
- ▶ Прерываемые сокеты
- ▶ Получение данных из Интернета
- ▶ Отправка электронной почты

Эта глава начинается с описания основных понятий для работы в сети, а затем в ней рассматриваются примеры написания программ на Java, позволяющих устанавливать соединения с серверами. Из нее вы узнаете, как осуществляется реализация сетевых клиентов и серверов. А завершается глава рассмотрением вопросов передачи почтовых сообщений из программы на Java и сбора данных с веб-сервера.

4.1. Подключение к серверу

В последующих разделах сначала рассматривается подключение к серверу вручную с помощью утилиты `telnet`, а затем автоматическое подключение из программы на Java.

4.1.1. Применение утилиты `telnet`

Утилита `telnet` служит отличным инструментальным средством для отладки сетевых программ. Она должна запускаться из командной строки по команде `telnet`.



На заметку! В Windows утилиту `telnet` необходимо активизировать. С этой целью откройте панель управления, перейдите в раздел Программы, щелкните на ссылке Добавление или удаление компонентов Windows и установите флашок Клиент Telnet. Следует также иметь в виду, что брандмауэр Windows блокирует некоторые сетевые порты, которые будут использоваться в примерах программ из этой главы. Чтобы разблокировать эти порты, вы должны обладать полномочиями администратора.

Утилитой `telnet` можно пользоваться не только для соединения с удаленным компьютером. С ее помощью можно также взаимодействовать с различными сетевыми службами. Ниже приводится один из примеров необычного использования этой утилиты. Для этого введите в командной строке следующую команду:

`telnet time-a.nist.gov 13`

На рис. 4.1 приведен пример ответной реакции сервера, которая в режиме командной строки будет иметь следующий вид:

54276 07-06-25 21:37:31 50 0 0 659.0 UTC(NIST) *

```

Terminal
$ telnet time-a.nist.gov 13
Trying 129.6.15.28...
Connected to time-a.nist.gov.
Escape character is '^'.
57488 16-04-10 04:23:00 50 0 0 610.5 UTC(NIST) *
Connection closed by foreign host.
-$

```

Рис. 4.1. Результат, получаемый из службы учета времени дня

Что же в действительности произошло? Утилита `telnet` подключилась к серверу службы учета времени дня, который работает на большинстве компьютеров под управлением операционной системы UNIX. Указанный в этом примере сервер находится в Национальном институте стандартов и технологий США (National Institute of Standards and Technology). Его системное время синхронизировано с цезиевыми атомными часами. (Безусловно, полученное значение текущего времени будет не совсем точным из-за задержек, связанных с передачей данных по сети.) По принятым правилам сервер службы времени всегда связан с портом 13.



На заметку! В сетевой терминологии порт – это не какое-то конкретное физическое устройство, а абстрактное понятие, упрощающее представление о соединении сервера с клиентом (рис. 4.2).

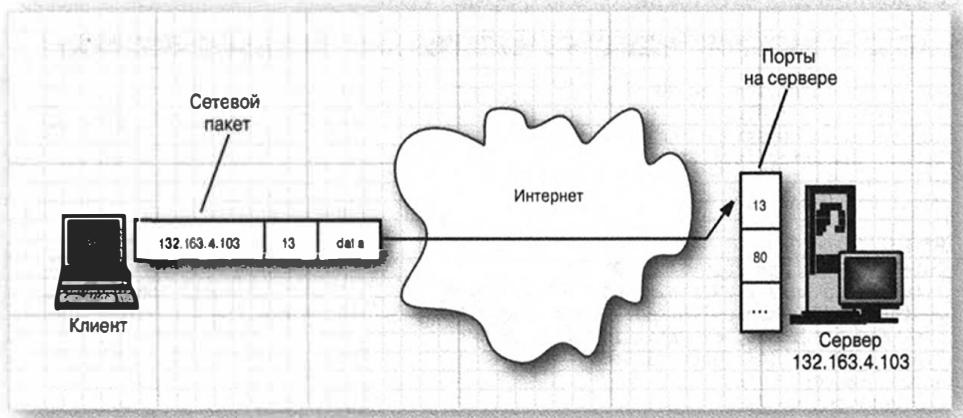


Рис. 4.2. Схема соединения клиента с сервером через конкретный порт

Программное обеспечение сервера постоянно работает на удаленном компьютере и ожидает поступления сетевого трафика через порт 13. При получении операционной системой на удаленном компьютере сетевого пакета с запросом на подключение к порту 13 на сервере активизируется соответствующий процесс и устанавливается соединение. Такое соединение может быть прервано одним из его участников.

Когда сеанс связи с сервером через порт 13 начинается по команде `telnet` с параметром `time-a.nist.gov`, сетевое программное обеспечение преобразует строку `"time-a.nist.gov"` в IP-адрес `129.6.15.28`. Затем оно посылает по этому адресу запрос на соединение с удаленным компьютером через порт 13. После установления соединения программа на удаленном компьютере передает обратно строку с данными, а затем разрывает соединение. Разумеется, клиенты и серверы могут вести и более сложные диалоги до разрыва соединения.

Проведем еще один, более интересный эксперимент. С этой целью выполните следующие действия.

1. Введите в режиме командной строки команду
`telnet horstmann.com 20`
2. Затем аккуратно и точно введите следующие строки, дважды нажав клавишу `<Enter>` в конце:

```
GET / HTTP/1.1
Host: horstmann.com
пустая строка
```

На рис. 4.3 показана ответная реакция сервера в окне утилиты telnet. Она имеет уже знакомый вам вид страницы текста в формате HTML, а именно начальной страницы веб-сайта Кея Хорстманна. Именно так обычный веб-браузер получает искомые веб-страницы. Для запроса веб-страниц на сервере он применяет сетевой протокол HTTP. Разумеется, браузер отображает данные в намного более удобном для чтения виде, чем формат HTML.



```

$ telnet horstmann.com 80
Trying 67.210.118.65...
Connected to horstmann.com.
Escape character is '^]'.
GET / HTTP/1.1
Host: horstmann.com

HTTP/1.1 200 OK
Date: Sun, 10 Apr 2016 04:36:27 GMT
Server: Apache/2.2.24 (Unix) mod_ssl/2.2.24 OpenSSL/0.9.8e-fips-rhel5 mod_auth_p
assthrough/2.1 mod_bwlimited/1.4 mod_fcgid/2.3.6 Sun-ONE-ASP/4.0.3
Last-Modified: Thu, 17 Mar 2016 18:32:18 GMT
ETag: "2590e1c1c47-52e42d9a8f680"
Accept-Ranges: bytes
Content-Length: 7239
Content-Type: text/html

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/x
html1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"><head>
  <title>Cay Horstmann's Home Page</title>
  <link href="styles.css" rel="stylesheet" type="text/css"/>
```

Рис. 4.3. Доступ к HTTP-порту с помощью утилиты telnet

 **На заметку!** Пару "ключ-значение" `Host: horstmann.com` требуется указывать для подключения к веб-серверу, на котором под одним и тем же IP-адресом размещаются разные домены. Ее можно не указывать, если на веб-сервере размещается единственный домен.

4.1.2. Подключение к серверу из программы на Java

В первом примере сетевой программы, исходный код которой приведен в листинге 4.1, выполняются те же самые действия, что и при использовании утилиты telnet. Она устанавливает соединение с сервером через порт и выводит получаемые в ответ данные.

Листинг 4.1. Исходный код из файла socket/SocketTest.java

```

1 package socket;
2
3 import java.io.*;
4 import java.net.*;
5 import java.util.*;
6
7 /**
8 * В этой программе устанавливается сокетное соединение
```

```
9  * с атомными часами в г. Боулдере, шт. Колорадо и выводится
10 * время, передаваемое из сервера
11 * @version 1.21 2016-04-27
12 * @author Cay Horstmann
13 */
14 public class SocketTest
15 {
16     public static void main(String[] args) throws IOException
17     {
18         try (Socket s = new Socket("time-a.nist.gov", 13);
19              Scanner in = new Scanner(s.getInputStream(), "UTF-8"))
20         {
21             while (in.hasNextLine())
22             {
23                 String line = in.nextLine();
24                 System.out.println(line);
25             }
26         }
27     }
28 }
```

В данной программе наибольший интерес представляют следующие две строчки кода:

```
Socket s = new Socket("time-a.nist.gov", 13);
Scanner in = new Scanner(s.getInputStream(), "UTF-8")
```

В первой строке кода открывается сокет. *Сокет* — это абстрактное понятие, обозначающее возможность для программ устанавливать соединения для обмена данными по сети. Конструктору объекта сокета передается адрес удаленного сервера и номер порта. Если установить соединение не удается, генерируется исключение типа `UnknownHostException`, а при возникновении каких-нибудь других затруднений — исключение типа `IOException`. Класс `UnknownHostException` является подклассом, производным от класса `IOException`, поэтому в данном простом примере обрабатывается только исключение из суперкласса.

После открытия сокета метод `getInputStream()` из класса `java.net.Socket` возвращает объект типа `InputStream`, который можно использовать как любой другой поток ввода. Получив поток ввода, рассматриваемая здесь программа приступает к выводу каждой введенной символьной строки в стандартный поток вывода. Этот процесс продолжается до тех пор, пока не завершится поток ввода или не разорвется соединение с сервером.

Данная программа может взаимодействовать только с очень простыми серверами, например, со службой учета текущего времени. В более сложных случаях клиент посыпает серверу запрос на получение данных, а сервер может поддерживать установленное соединение в течение некоторого времени после отправки ответа на запрос. Примеры реализации подобного поведения представлены далее в этой главе.

Класс `Socket` очень удобен для работы в сети, поскольку он скрывает все сложности и подробности установления сетевого соединения и передачи данных по сети, реализуемые средствами библиотеки Java. А пакет `java.net`, по существу, предоставляет тот же самый программный интерфейс, который используется для работы с файлами.



На заметку! Здесь рассматривается только сетевой протокол TCP [Transmission Control Protocol — протокол управления передачей]. На платформе Java поддерживается также протокол UDP (User Datagram Protocol — протокол пользовательских дейтаграмм), который может служить для отправки пакетов (называемых иначе дейтаграммами) с гораздо меньшими издержками, чем по протоколу TCP. Недостаток такого способа обмена данными по сети заключается в том, что пакеты необязательно доставлять получателю в последовательном порядке, и они вообще могут быть потеряны. Получатель сам должен позаботиться о том, чтобы пакеты были организованы в определенном порядке, а кроме того, он должен сам запрашивать повторную передачу отсутствующих пакетов. Протокол UDP хорошо подходит для тех приложений, которые могут обходиться без отсутствующих пакетов, например, для организации аудио- и видеопотоков или продолжительных измерений.

java.net.Socket 1.0

- **Socket(String host, int port)**
 - Создает сокет для соединения с указанным хостом или портом.
 - **InputStream getInputStream()**
 - **OutputStream getOutputStream()**
- Получают поток ввода для чтения данных из сокета или поток вывода для записи данных в сокет.

4.1.3. Время ожидания для сокетов

Чтение данных из сокета продолжается до тех пор, пока данные доступны. Если хост (т.е. сетевой узел) недоступен, прикладная программа будет ожидать очень долго, и все будет зависеть от того, когда операционная система, под управлением которой работает компьютер, определит момент завершения времени ожидания.

Для конкретной прикладной программы можно самостоятельно определить наиболее подходящую величину времени ожидания для сокета, а затем вызвать метод `setSoTimeout()`, чтобы установить эту величину в миллисекундах. В приведенном ниже фрагменте кода показано, как это делается.

```
Socket s = new Socket(.. . .);
s.setSoTimeout(10000); // истечение времени ожидания через 10 секунд
```

Если величина времени ожидания была задана для сокета, то при выполнении всех последующих операций чтения и записи данных будет генерироваться исключение типа `SocketTimeoutException` по истечении времени ожидания до фактического завершения текущей операции. Но это исключение можно перехватить, чтобы отреагировать на данное событие надлежащим образом, как показано ниже.

```
try
{
    InputStream in = s.getInputStream();
    // читать данные из потока ввода in
    ...
}
catch (InterruptedIOException exception)
```

```
{  
    отреагировать на истечение времени ожидания  
}
```

Что касается времени ожидания для сокетов, то остается еще одно затруднение, которое придется каким-то образом разрешить. Так, приведенный ниже конструктор может установить блокировку в течение неопределенного периода времени до тех пор, пока не будет установлено первоначальное соединение с хостом.

```
Socket(String host, int port)
```

Это затруднение можно преодолеть, если сначала создать несоединяемый сокет, а затем установить соединение с ним, задав время ожидания:

```
Socket s = new Socket();  
s.connect(new InetSocketAddress(host, port), timeout);
```

Если же пользователям требуется предоставить возможность прерывать соединение с сокетом в любой момент, то далее, в разделе 4.3, поясняется, как этого добиться.

java.net.Socket 1.0

- **Socket() 1.1**
Создает сокет, который еще не соединен в данный момент времени.
- **void connect(SocketAddress address) 1.4**
Соединяет данный сокет по указанному адресу.
- **void connect(SocketAddress address, int timeoutInMilliseconds) 1.4**
Соединяет данный сокет по указанному адресу или осуществляет возврат, если заданный промежуток времени истек.
- **void setSoTimeout(int timeoutInMilliseconds) 1.1**
Задает время блокировки для чтения запросов в данном сокете. По истечении времени блокировки возникает исключение типа `InterruptedException`.
- **boolean isConnected() 1.4**
Возвращает логическое значение `true`, если установлено соединение с сокетом.
- **boolean isClosed() 1.4**
Возвращает логическое значение `true`, если разорвано соединение с сокетом.

4.1.4. Межсетевые адреса

Как правило, нет особой нужды беспокоиться о межсетевых адресах в Интернете — числовых адресах хостов, состоящих из четырех байтов (или из шестнадцати — по протоколу IPv6), как, например, `129.6.15.28`. Тем не менее, если требуется выполнить взаимное преобразование имен хостов и межсетевых адресов, то для этой цели можно воспользоваться классом `InetAddress`.

В пакете `java.net` поддерживаются межсетевые адреса по протоколу IPv6, при условии, что их поддержка обеспечивается и со стороны операционной системы хоста. В частности, статический метод `getByName()` возвращает объект типа `InetAddress` для хоста. Например, в следующей строке кода возвращается

объект типа `InetAddress`, инкапсулирующий последовательность из четырех байтов **129.6.15.28**:

```
InetAddress address = InetAddress.getByName("time-a.nist.gov");
```

Чтобы получить байты межсетевого адреса, достаточно вызвать метод `getAddress()` следующим образом:

```
byte[] addressBytes = address.getAddress();
```

Имена некоторых хостов с большим объемом трафика соответствуют нескольким межсетевым адресам, что объясняется попыткой сбалансировать нагрузку. Так, на момент написания данной книги имя хоста `google.com` соответствовало двенадцати сетевым адресам. Один из них выбирается случайным образом во время доступа к хосту. Получить межсетевые адреса всех хостов можно, вызвав метод `getAllByName()`:

```
InetAddress[] addresses = InetAddress.getAllByName(host);
```

И, наконец, иногда требуется адрес локального хоста. Если вы просто запросите адрес локального хоста, указав `localhost`, то неизменно получите в ответ локальный петлевой адрес `127.0.0.1`, которым другие не смогут воспользоваться для подключения к вашему компьютеру. Вместо этого вызовите метод `getLocalHost()`, чтобы получить адрес вашего локального хоста, как показано ниже.

```
InetAddress address = InetAddress.getLocalHost();
```

В листинге 4.2 приведен пример простой программы, выводящей межсетевой адрес локального хоста, если не указать дополнительные параметры в командной строке, или же все межсетевые адреса другого хоста, если указать имя хоста в командной строке, как в следующем примере:

```
java InetAddress/InetAddressTest www.horstmann.com
```

Листинг 4.2. Исходный код из файла `inetAddress/InetAddressTest.java`

```

1 package inetAddress;
2
3 import java.io.*;
4 import java.net.*;
5 /**
6  * В этой программе демонстрируется применение класса InetAddress.
7  * В качестве аргумента в командной строке следует указать имя
8  * хоста или запустить программу без аргументов, чтобы получить
9  * в ответ адрес локального хоста
10 * @version 1.02 2012-06-05
11 * @author Cay Horstmann
12 */
13 public class InetAddressTest
14 {
15     public static void main(String[] args) throws IOException
16     {
17         if (args.length > 0)
18         {
19             String host = args[0];
20             InetAddress[] addresses = InetAddress.getAllByName(host);
21             for (InetAddress a : addresses)
22             {
23                 System.out.println(a);
24             }
25         }
26     }
27 }
```

```
22     System.out.println(a);
23 }
24 else
25 {
26     InetAddress localHostAddress = InetAddress.getLocalHost();
27     System.out.println(localHostAddress);
28 }
29 }
30 }
```

java.net.InetAddress 1.0

- **static InetAddress getByName(String host)**
Конструирует объект типа `InetAddress` или массив всех межсетевых адресов для заданного имени хоста.
- **static InetAddress getAllByName(String host)**
Конструирует объект типа `InetAddress` для локального хоста.
- **byte[] getAddress()**
Возвращает массив байтов, содержащий числовой адрес.
- **String getHostAddress()**
Возвращает адрес хоста в виде символьной строки с десятичными числами, разделенными точками, например "132.163.4.102".
- **String getHostName()**
Возвращает имя хоста.

4.2. Реализация серверов

В предыдущем разделе были рассмотрены особенности реализации элементарного сетевого клиента, способного получать данные из сети. А теперь переходим к обсуждению реализации простого сервера, способного посыпать данные клиентам.

4.2.1. Сокеты сервера

После запуска серверная программа переходит в режим ожидания от клиентов подключения к портам сервера. Для рассматриваемого здесь примера выбран номер порта **8189**, который не используется ни одной из стандартных служб. В следующей строке кода создается сервер с контролируемым портом **8189**:

```
ServerSocket s = new ServerSocket(8189);
```

А в приведенной ниже строке кода серверной программе предписывается ожидать подключения клиентов к заданному порту.

```
Socket incoming = s.accept();
```

Как только какой-нибудь клиент подключится к данному порту, отправив по сети запрос на сервер, метод `accept()` возвратит объект типа `Socket`,

представляющий установленное соединение. Этот объект можно использовать для чтения и записи данных в потоки ввода-вывода, как показано в приведенном ниже фрагменте кода.

```
InputStream inStream = incoming.getInputStream();
OutputStream outStream = incoming.getOutputStream();
```

Все данные, направляемые в поток вывода серверной программы, поступают в поток ввода клиентской программы. А все данные, направляемые в поток вывода из клиентской программы, поступают в поток ввода серверной программы. Во всех примерах, приведенных в этой главе, обмен текстовыми данными осуществляется через сокеты. Поэтому соответствующие потоки ввода-вывода через сокет преобразуются в потоки сканирования (типа Scanner) и записи (типа Writer) следующим образом:

```
Scanner in = new Scanner(inStream, "UTF-8");
PrintWriter out = new PrintWriter(new OutputStreamWriter(
    outStream, "UTF-8"), true /* автоматическая очистка */);
```

Допустим, клиентская программа посыпает следующее приветствие:

```
out.println("Hello! Enter BYE to exit.");
```

Если для подключения к серверной программе через порт **8189** используется утилита telnet, это приветствие отображается на экране терминала.

В рассматриваемой здесь простой серверной программе вводимые данные, отправленные клиентской программой,читываются построчно и посыпаются обратно клиентской программе в режиме эхопередачи, как показано в приведенном ниже фрагменте кода. Этим наглядно демонстрируется получение данных от клиентской программы. А настоящая серверная программа должна обрабатывать полученные данные и выдать соответствующий ответ.

```
String line = in.nextLine();
out.println("Echo: " + line);
if (line.trim().equals("BYE")) done = true;
```

По завершении сеанса связи открытый сокет закрывается следующим образом:

```
incoming.close();
```

Вот, собственно, и все, что делает данная программа. Любая серверная программа, например, веб-сервер, работающий по протоколу HTTP, выполняет аналогичный цикл следующих действий.

1. Получение из потока ввода входящих данных запроса на конкретную информацию от клиентской программы.
2. Расшифровка клиентского запроса.
3. Сбор информации, запрашиваемой клиентом.
4. Передача обнаруженной информации клиентской программе через поток вывода исходящих данных.

В листинге 4.3 приведен весь исходный код описанного выше примера серверной программы.

Листинг 4.3. Исходный код из файла server/EchoServer.java

```
1 package server;
2
3 import java.io.*;
4 import java.net.*;
5 import java.util.*;
6 /**
7  * В этой программе реализуется простой сервер,
8  * прослушивающий порт 8189 и посылающий обратно
9  * клиенту все полученные от него данные
10 * @version 1.21 2012-05-19
11 * @author Cay Horstmann
12 */
13 public class EchoServer
14 {
15     public static void main(String[] args) throws IOException
16     {
17         // установить сокет на стороне сервера
18         try (ServerSocket s = new ServerSocket(8189))
19         {
20             // ожидать подключения клиента
21             try (Socket incoming = s.accept())
22             {
23                 InputStream inStream = incoming.getInputStream();
24                 OutputStream outStream = incoming.getOutputStream();
25
26                 try (Scanner in = new Scanner(inStream, "UTF-8"))
27                 {
28                     PrintWriter out = new PrintWriter(
29                         new OutputStreamWriter(outStream, "UTF-8"),
30                         true /* автоматическая очистка */);
31
32                     out.println("Hello! Enter BYE to exit.");
33
34                     // передать обратно данные, полученные от клиента
35                     boolean done = false;
36                     while (!done && in.hasNextLine())
37                     {
38                         String line = in.nextLine();
39                         out.println("Echo: " + line);
40                         if (line.trim().equals("BYE")) done = true;
41                     }
42                 }
43             }
44         }
45     }
46 }
```

Для проверки работоспособности данной серверной программы ее нужно скомпилировать и запустить. Затем необходимо подключиться с помощью утилиты telnet к локальному серверу localhost (или по IP-адресу 127.0.0.1) через порт **8189**. Если ваш компьютер непосредственно подключен к Интернету, любой пользователь может получить доступ к данной серверной программе, если ему известен IP-адрес и номер порта. При подключении через этот порт будет получено следующее сообщение (рис. 4.4):

Hello! Enter BYE to exit.

(Привет! Введите BYE (Пока), чтобы выйти из программы)

```

Terminal
File Edit View Terminal Tabs Help
~$ telnet localhost 8189
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello! Enter BYE to exit.
Hello Sailor!
Echo: Hello Sailor!
BYE
Echo: BYE
Connection closed by foreign host.
~$ █

```

Рис. 4.4. Сеанс связи с сервером, передающим обратно данные, полученные от клиента

Введите любую фразу и понаблюдайте за тем, как она будет получена обратно в том же самом виде. Для отключения от сервера введите **BYE** (все символы в верхнем регистре). В итоге завершится и серверная программа.

java.net.ServerSocket 1.0

- **ServerSocket(int port)**
Создает сокет на стороне сервера, контролирующего указанный порт.
- **Socket accept()**
Ожидает соединения. Этот метод блокирует (т.е. переводит в режим ожидания) текущий поток до тех пор, пока не будет установлено соединение. Возвращает объект типа **Socket**, через который программа может взаимодействовать с подключаемым клиентом.
- **void close()**
Закрывает сокет на стороне сервера.

4.2.2. Обслуживание многих клиентов

В предыдущем простом примере серверной программы не предусмотрена возможность одновременного подключения сразу нескольких клиентских программ. Обычно серверная программа работает на компьютере сервера, а клиентские программы могут одновременно подключаться к ней через Интернет из любой точки мира. Если на сервере не предусмотрена обработка одновременных запросов от многих клиентов, один из клиентов может монополизировать доступ

к серверной программе в течение длительного времени. Во избежание подобных ситуаций следует прибегнуть к помощи потоков исполнения.

Всякий раз, когда серверная программа устанавливает новое сокетное соединение, т.е. в результате вызова метода `accept()` возвращается сокет, запускается новый поток исполнения для подключения *данного* клиента к серверу. После этого происходит возврат в основную программу, которая переходит в режим ожидания следующего соединения. Для того чтобы все это произошло, в серверной программе следует организовать приведенный ниже основной цикл.

```
while (true)
{
    Socket incoming = s.accept();
    Runnable r = new ThreadedEchoHandler(incoming);

    Thread t = new Thread(r);
    t.start();
}
```

Класс `ThreadedEchoHandler` реализует интерфейс `Runnable` и в своем методе `run()` поддерживает взаимодействие с клиентской программой:

```
class ThreadedEchoHandler implements Runnable
{
    ...
    public void run()
    {
        try (InputStream inStream = incoming.getInputStream();
            OutputStream outStream = incoming.getOutputStream())
        {
            обработать полученный запрос и отправить ответ
        }
        catch (IOException e)
        {
            обработать исключение
        }
    }
}
```

Когда запускается новый поток исполнения, при каждом соединении, несколько клиентских программ могут одновременно подключаться к серверу. Это нетрудно проверить, выполнив следующие действия.

1. Скомпилируйте и запустите на выполнение серверную программу, исходный код которой приведен в листинге 4.4.
2. Откройте несколько окон утилиты `telnet` (рис. 4.5).
3. Переходя из одного окна в другое, введите команды. В итоге каждое отдельное окно утилиты `telnet` будет взаимодействовать с серверной программой независимо от других окон.
4. Чтобы разорвать соединение и закрыть окно утилиты `telnet`, нажмите комбинацию клавиш `<Ctrl+C>`.

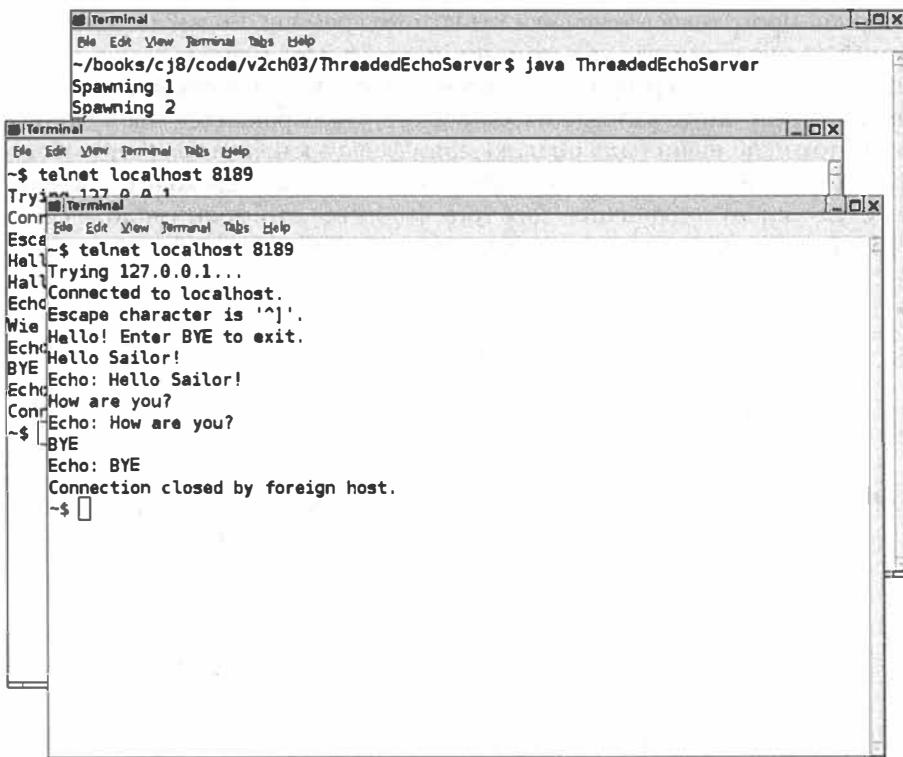


Рис. 4.5. Сеанс одновременной связи нескольких клиентов с сервером

На заметку! В рассматриваемой здесь программе для каждого соединения порождается отдельный поток исполнения. Такой прием не вполне подходит для высокопроизводительного сервера. Более эффективной работы сервера можно добиться, используя средства из пакета `java.nio`. Дополнительные сведения по данному вопросу можно получить, обратившись по адресу <http://www.ibm.com/developerworks/library/j-jav aio/>.

Листинг 4.4. Исходный код из файла `threaded/ThreadedEchoServer.java`

```

1 package threaded;
2
3 import java.io.*;
4 import java.net.*;
5 import java.util.*;
6
7 /**
8  * В этой программе реализуется многопоточный сервер,
9  * прослушивающий порт 8189 и передающий обратно все данные,
10 * полученные от клиентов
11 * all client input.
12 * @author Cay Horstmann
13 * @version 1.22 2016-04-27
14 */
15 public class ThreadedEchoServer

```

```
16 {
17     public static void main(String[] args )
18     {
19         try (ServerSocket s = new ServerSocket(8189))
20         {
21             int i = 1;
22
23             while (true)
24             {
25                 Socket incoming = s.accept();
26                 System.out.println("Spawning " + i);
27                 Runnable r = new ThreadedEchoHandler(incoming);
28                 Thread t = new Thread(r);
29                 t.start();
30                 i++;
31             }
32         }
33         catch (IOException e)
34         {
35             e.printStackTrace();
36         }
37     }
38 }
39
40 /**
41  * Этот класс обрабатывает данные, получаемые сервером от
42  * клиента через одно сокетное соединение
43 */
44 class ThreadedEchoHandler implements Runnable
45 {
46     private Socket incoming;
47
48     /**
49      Конструирует обработчик
50      @param incomingSocket Входящий сокет
51     */
52     public ThreadedEchoHandler(Socket incomingSocket)
53     {
54         incoming = incomingSocket;
55     }
56
57     public void run()
58     {
59         try (InputStream inStream = incoming.getInputStream();
60              OutputStream outStream = incoming.getOutputStream())
61         {
62             Scanner in = new Scanner(inStream, "UTF-8");
63             PrintWriter out = new PrintWriter(
64                 new OutputStreamWriter(outStream, "UTF-8"),
65                 true /* автоматическая очистка */);
66
67             out.println("Hello! Enter BYE to exit.");
68
69             // передать обратно данные, полученные от клиента
70             boolean done = false;
71             while (!done && in.hasNextLine())
72             {
73                 String line = in.nextLine();
74                 out.println("Echo: " + line);
75                 if (line.trim().equals("BYE"))
```

```

76         done = true;
77     }
78 }
79 catch (IOException e)
80 {
81     e.printStackTrace();
82 }
83 }
84 }
```

4.2.3. Полузакрытие

Полузакрытие обеспечивает возможность прервать передачу данных на одной стороне сокетного соединения, продолжая в то же время прием данных от другой стороны. Рассмотрим типичную ситуацию. Допустим, данные направляются на сервер, но заранее неизвестно, какой именно объем данных требуется передать. Если речь идет о файле, то его закрытие, по существу, означает завершение передачи данных. Если же закрыть сокет, то соединение с сервером будет немедленно разорвано.

Преодолеть подобное затруднение призвано полузакрытие. Если закрыть поток вывода через сокет, то для сервера это будет означать завершение передачи данных запроса. При этом поток ввода остается открытym, позволяя получить ответ от сервера. Код, реализующий механизм полузакрытия на стороне клиента, приведен ниже.

```

try (Socket socket = new Socket(host, port))
{
    Scanner in = new Scanner(socket.getInputStream(), "UTF-8");
    PrintWriter writer = new PrintWriter(socket.getOutputStream());
    // передать данные запроса
    writer.print(..);
    writer.flush();
    socket.shutdownOutput();
    // теперь сокет полузакрыт
    // принять данные ответа
    while (in.hasNextLine() != null)
        { String line = in.nextLine(); ... }
}
```

Серверная программа просто читает данные из потока ввода до тех пор, пока не закроется поток вывода на другом конце соединения. Очевидно, что такой подход применим только для служб однократного действия по сетевым протоколам, подобным HTTP, где клиент устанавливает соединение с сервером, передает запрос, получает ответ, после чего соединение разрывается.

java.net.Socket 1.0

- **void shutdownOutput() 1.3**
Устанавливает поток вывода в состояние завершения.
- **void shutdownInput() 1.3**
Устанавливает поток ввода в состояние завершения.

java.net.Socket 1.0 (окончание)**• boolean isOutputShutdown() 1.4**

Возвращает логическое значение `true`, если вывод данных был остановлен.

• boolean isInputShutdown() 1.4

Возвращает логическое значение `true`, если ввод данных был остановлен.

4.3. Прерываемые сокеты

При подключении через сокет текущий поток исполнения блокируется до тех пор, пока соединение не будет установлено, или же до истечения времени ожидания. Аналогично, если пытаться передать или принять данные через сокет, текущий поток приостановит свое исполнение до успешного завершения операции или до истечения времени ожидания.

В прикладных программах, работающих в диалоговом режиме, пользователям желательно предоставить возможность прервать слишком затянувшийся процесс установления соединения через сокет. Но если поток исполнения блокирован для нереагирующего сокета, то разблокировать его не удастся, вызвав метод `interrupt()`.

Для прерывания сокетных операций служит класс `SocketChannel`, предоставляемый в пакете `java.nio`. Объект типа `SocketChannel` создается следующим образом:

```
SocketChannel channel =
    SocketChannel.open(new InetSocketAddress(host, port));
```

У канала отсутствуют связанные с ним потоки ввода-вывода. Вместо этого в канале предоставляются методы `read()` и `write()`, использующие объекты типа `Buffer`. (Подробнее о буферах см. в главе 2.) Эти методы объявляются в интерфейсах `ReadableByteChannel` и `WritableByteChannel`. Если же нет желания иметь дело с буферами, для чтения из канала типа `SocketChannel` можно воспользоваться объектом типа `Scanner`. Для этой цели в классе `Scanner` предусмотрен следующий конструктор с параметром типа `ReadableByteChannel`:

```
Scanner in = new Scanner(channel, "UTF-8");
```

Чтобы превратить канал в поток вывода, применяется статический метод `Channels.newOutputStream()`:

```
OutputStream outStream = Channels.newOutputStream(channel);
```

Вот, собственно, и все, что нужно сделать для прерывания сокетной операции. Если же поток исполнения будет прерван в процессе установления соединения, чтения или записи, соответствующая операция завершится генерированием исключения.

В примере программы, исходный код которой приведен в листинге 4.5, демонстрируется применение прерываемых и блокирующих сокетов. Сервер передает числовые данные, имитируя прерывание их передачи после десятого числа. Если щелкнуть на любой кнопке, запустится поток исполнения, устанавливающий

соединение с сервером и выводящий на экран передаваемые данные. В первом потоке исполнения используется прерываемый сокет, а во втором — блокирующий. Если щелкнуть на кнопке Cancel (Отмена) во время вывода первых десяти чисел, то прервется исполнение любого из двух потоков.

А если щелкнуть на кнопке Cancel после передачи первых десяти чисел, то прервется исполнение только первого потока. Блокировка второго потока исполнения будет продолжаться до тех пор, пока сервер окончательно не разорвет соединение (рис. 4.6).

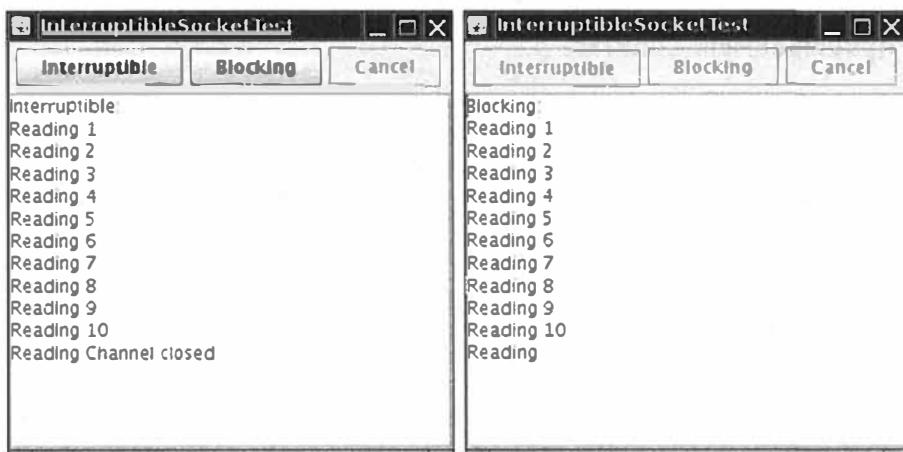


Рис. 4.6. Прерывание сокета

Листинг 4.5. Исходный код из файла interruptible/InterruptibleSocketTest.java

```

1 package interruptible;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.util.*;
6 import java.net.*;
7 import java.io.*;
8 import java.nio.channels.*;
9 import javax.swing.*;
10
11 /**
12 * В этой программе демонстрируется прерывание сокета через канал
13 * @author Cay Horstmann
14 * @version 1.04 2016-04-27
15 */
16 public class InterruptibleSocketTest
17 {
18     public static void main(String[] args)
19     {
20         EventQueue.invokeLater(() ->
21         {
22             JFrame frame = new InterruptibleSocketFrame();
23             frame.setTitle("InterruptibleSocketTest");
24             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

```

```
25         frame.setVisible(true);
26     });
27 }
28 }
29
30 class InterruptibleSocketFrame extends JFrame
31 {
32     private Scanner in;
33     private JButton interruptibleButton;
34     private JButton blockingButton;
35     private JButton cancelButton;
36     private JTextArea messages;
37     private TestServer server;
38     private Thread connectThread;
39
40     public InterruptibleSocketFrame()
41     {
42         JPanel northPanel = new JPanel();
43         add(northPanel, BorderLayout.NORTH);
44
45         final int TEXT_ROWS = 20;
46         final int TEXT_COLUMNS = 60;
47         messages = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
48         add(new JScrollPane(messages));
49
50         interruptibleButton = new JButton("Interruptible");
51         blockingButton = new JButton("Blocking");
52
53         northPanel.add(interruptibleButton);
54         northPanel.add(blockingButton);
55
56         interruptibleButton.addActionListener(event ->
57         {
58             interruptibleButton.setEnabled(false);
59             blockingButton.setEnabled(false);
60             cancelButton.setEnabled(true);
61             connectThread = new Thread(() ->
62             {
63                 try
64                 {
65                     connectInterruptibly();
66                 }
67                 catch (IOException e)
68                 {
69                     messages.append("\nInterruptibleSocketTest
70                         .connectInterruptibly: " + e);
71                 }
72             });
73             connectThread.start();
74         });
75
76         blockingButton.addActionListener(event ->
77         {
78             interruptibleButton.setEnabled(false);
79             blockingButton.setEnabled(false);
80             cancelButton.setEnabled(true);
81             connectThread = new Thread(() ->
82             {
83                 try
```

```
84         {
85             connectBlocking();
86         }
87     catch (IOException e)
88     {
89         messages.append("\nInterruptibleSocketTest
90             .connectBlocking: " + e);
90     }
91 }
92     connectThread.start();
93 });
94 );
95
96 cancelButton = new JButton("Cancel");
97 cancelButton.setEnabled(false);
98 northPanel.add(cancelButton);
99 cancelButton.addActionListener(event ->
100     {
101         connectThread.interrupt();
102         cancelButton.setEnabled(false);
103     });
104 server = new TestServer();
105 new Thread(server).start();
106 pack();
107 }
108
109 /**
110 * Соединяет с проверяемым сервером,
111 * используя прерываемый ввод-вывод
112 */
113 public void connectInterruptibly() throws IOException
114 {
115     messages.append("Interruptible:\n");
116     try (SocketChannel channel = SocketChannel.open(
117         new InetSocketAddress("localhost", 8189)))
118     {
119         in = new Scanner(channel, "UTF-8");
120         while (!Thread.currentThread().isInterrupted())
121         {
122             messages.append("Reading ");
123             if (in.hasNextLine())
124             {
125                 String line = in.nextLine();
126                 messages.append(line);
127                 messages.append("\n");
128             }
129         }
130     }
131     finally
132     {
133         EventQueue.invokeLater(() ->
134         {
135             messages.append("Channel closed\n");
136             interruptibleButton.setEnabled(true);
137             blockingButton.setEnabled(true);
138         });
139     }
140 }
141 /**
142 */
```

```
143 * Соединяет с проверяемым сервером,
144 * используя блокирующий ввод-вывод
145 */
146 public void connectBlocking() throws IOException
147 {
148     messages.append("Blocking:\n");
149     try (Socket sock = new Socket("localhost", 8189))
150     {
151         in = new Scanner(sock.getInputStream(), "UTF-8");
152         while (!Thread.currentThread().isInterrupted())
153         {
154             messages.append("Reading ");
155             if (in.hasNextLine())
156             {
157                 String line = in.nextLine();
158                 messages.append(line);
159                 messages.append("\n");
160             }
161         }
162     }
163     finally
164     {
165         EventQueue.invokeLater(() ->
166         {
167             messages.append("Socket closed\n");
168             interruptibleButton.setEnabled(true);
169             blockingButton.setEnabled(true);
170         });
171     }
172 }
173 /**
174 * Многопоточный сервер, прослушивающий порт 8189 и посылающий
175 * клиентам числа, имитируя зависание после передачи 10 чисел
176 */
177 class TestServer implements Runnable
178 {
179     public void run()
180     {
181         try (ServerSocket s = new ServerSocket(8189))
182         {
183             while (true)
184             {
185                 Socket incoming = s.accept();
186                 Runnable r = new TestServerHandler(incoming);
187                 Thread t = new Thread(r);
188                 t.start();
189             }
190         }
191     }
192     catch (IOException e)
193     {
194         messages.append("\nTestServer.run: " + e);
195     }
196 }
197 /**
198 * Этот класс обрабатывает данные, получаемые сервером
199 * от клиента через одно сокетное соединение
200 *
```

```

202     */
203     class TestServerHandler implements Runnable
204     {
205         private Socket incoming;
206         private int counter;
207
208         /**
209          * Конструирует обработчик
210          * @param i входящий сокет
211          */
212         public TestServerHandler(Socket i)
213         {
214             incoming = i;
215         }
216
217         public void run()
218         {
219             try
220             {
221                 try
222                 {
223                     OutputStream outStream = incoming.getOutputStream();
224                     PrintWriter out = new PrintWriter(
225                         new OutputStreamWriter(outStream, "UTF-8"),
226                         true /* автоматическая очистка */);
227                     while (counter < 100)
228                     {
229                         counter++;
230                         if (counter <= 10) out.println(counter);
231                         Thread.sleep(100);
232                     }
233                 }
234                 finally
235                 {
236                     incoming.close();
237                     messages.append("Closing server\n");
238                 }
239             }
240             catch (Exception e)
241             {
242                 messages.append("\nTestServerHandler.run: " + e);
243             }
244         }
245     }
246 }
```

java.net.InetSocketAddress 1.4

- **InetSocketAddress(String hostname, int port)**

Создает объект адреса с указанными именем хоста (т.е. сетевого узла) и номером порта, преобразуя имя узла в адрес при установлении соединения. Если не удается преобразовать имя хоста в адрес, устанавливается логическое значение **true** свойства **unresolved**.

- **boolean isUnresolved()**

Возвращает логическое значение **true**, если для данного объекта не удается преобразовать имя хоста в адрес.

java.nio.channels.SocketChannel 1.4

- **static SocketChannel open(SocketAddress address)**

Открывает канал для сокета и связывает его с удаленным хостом по указанному адресу.

java.nio.channels.Channels 1.4

- **static InputStream newInputStream(ReadableByteChannel channel)**

Создает поток ввода для чтения данных из указанного канала.

- **static OutputStream newOutputStream(WritableByteChannel channel)**

Создает поток вывода для записи данных в указанный канал.

4.4. Получение данных из Интернета

Чтобы получить доступ к веб-серверам из программы на Java, требуется более высокий уровень сетевого взаимодействия, чем установление соединения через сокет и выдача HTTP-запросов. В последующих разделах будут рассмотрены классы, предоставляемые для этой цели в библиотеке Java.

4.4.1. URL и URI

Классы URL и URLConnection инкапсулируют большую часть внутреннего механизма извлечения данных с удаленного веб-сайта. Объект типа URL создается следующим образом:

```
URL url = new URL(символьная строка с URL);
```

Если требуется только извлечь содержимое из указанного ресурса, достаточно вызвать метод `openStream()` из класса URL. Этот метод возвращает объект типа `InputStream`. Поток ввода данного типа можно использовать обычным образом, например, создать объект типа `Scanner`:

```
InputStream inStream = url.openStream();
Scanner in = new Scanner(inStream, "UTF-8");
```

В пакете `java.net` отчетливо различаются унифицированные *указатели* ресурсов (URL) и унифицированные *идентификаторы* ресурсов (URI). В частности, URI – это лишь синтаксическая конструкция, содержащая различные части символьной строки, обозначающей веб-ресурс. А URL – это особая разновидность идентификатора URI с исчерпывающими данными о местоположении ресурса. Имеются и такие URI, как, например, `mailto:cay@hortsmann.com`, которые не являются указателями ресурсов, потому что по ним нельзя обнаружить какие-нибудь данные. Такой URI называется *унифицированным именем* ресурса (URN).

В классе URI из библиотеки Java отсутствуют методы доступа к ресурсу по указанному идентификатору, поскольку этот класс предназначен только для синтаксического анализа символьной строки, обозначающей ресурс. В отличие от него, класс URL позволяет открыть поток ввода-вывода для данного ресурса. Поэтому

в классе URL допускается взаимодействие только по тем протоколам и схемам, которые поддерживаются в библиотеке Java, в том числе http:, https: и ftp: — для Интернета, file: — для локальной файловой системы, а также jar: — для обращения к архивным JAR-файлам.

Синтаксический анализ URI — непростая задача, поскольку идентификаторы ресурсов могут иметь сложную структуру. В качестве примера ниже приведены URI с замысловатой структурой.

```
http://google.com?q=Beach+Chalet  
ftp://username:password@ftp.yourserver.com/pub/file.txt
```

В обозначении URI задаются правила создания таких идентификаторов. Структура URI выглядит следующим образом:

```
[схема:] специальная_часть_схемы[#фрагмент]
```

где квадратные скобки ([]) обозначают необязательную часть, а двоеточие и знак # служат в качестве разделителей. Если схема: является составной частью идентификатора URI, то он называется *абсолютным*, а иначе — *относительным*. Абсолютный URI называется *непрозрачным*, если специальная_часть_схемы не начинается с косой черты (/), как, например, показано ниже.

```
mailto:cay@horstmann.com
```

Все абсолютные, непрозрачные и относительные URL имеют *иерархическую структуру*. Ниже приведены характерные тому примеры.

```
http://horstmann.com/index.html  
.../java/net/Socket.html#Socket()
```

Составляющая специальная_часть_схемы иерархического URI имеет следующую структуру:

```
[//полномочия] [путь] [?запрос]
```

И здесь квадратные скобки ([]) обозначают необязательную часть. Составляющая *полномочия* в URI серверов имеет приведенную ниже форму, где элемент *порт* должен иметь целочисленное значение.

```
[сведения_о_пользователе@] хост[:порт]
```

В документе RFC 2396, стандартизирующем идентификаторы URI, допускается также механизм указания составляющей *полномочия* в другом формате на основе данных из реестра. Но он не получил широкого распространения.

Одно из назначений класса URI состоит в синтаксическом анализе отдельных составляющих идентификатора. Они извлекаются с помощью перечисленных ниже методов.

```
GetScheme()  
getSchemeSpecificPart()  
getAuthority()  
getUserInfo()  
getHost()  
getPort()  
getPath()  
getQuery()  
getFragment()
```

Другое назначение класса URI состоит в обработке абсолютных и относительных идентификаторов. Так, если имеются абсолютный и относительный идентификаторы URI:

`http://docs.mycompany.com/api/java/net/ServerSocket.html`

и

`../../java/net/Socket.html#Socket()`

их можно объединить в абсолютный URI следующим образом:

`http://docs.mycompany.com/api/java/net/Socket.html#Socket()`

Такой процесс называется *преобразованием адресов относительного URI*. Обратный процесс называется *преобразованием абсолютных адресов в относительные*. Например, имея базовый URI:

`http://docs.mycompany.com/api`

можно преобразовать следующий абсолютный URI:

`http://docs.company.com/api/java/lang/String.html`

в приведенный ниже относительный URI.

`java/lang/String.html`

Для выполнения обоих видов преобразования в классе URI предусмотрены два соответствующих метода:

```
relative = base.relativize(combined);  
combined = base.resolve(relative);
```

4.4.2. Извлечение данных средствами класса URLConnection

Для получения дополнительных сведений о веб-ресурсе следует воспользоваться классом URLConnection, предоставляющим намного больше средств управления доступом к веб-ресурсам, чем более простой класс URL. Для работы с объектом типа URLConnection необходимо тщательно спланировать и выполнить следующие действия.

1. Вызывать метод `openConnection()` из класса URL для получения объекта типа URLConnection следующим образом:

```
URLConnection connection = url.openConnection();
```

2. Задать свойства запроса с помощью перечисленных ниже методов.

```
SetDoInput()  
setDoOutput()  
setIfModifiedSince()  
setUseCaches()  
setAllowUserInteraction()  
setRequestProperty()  
setConnectTimeout()  
setReadTimeout()
```

3. Эти методы будут подробно рассматриваться далее.

4. Установить соединение с удаленным ресурсом с помощью метода `connect()`:
`connection.connect();`

5. Помимо создания сокета, для установления соединения с веб-сервером этот метод запрашивает также у сервера *данные заголовка*.
6. После подключения к веб-серверу становятся доступными поля заголовка. Обращаться к ним можно с помощью универсальных методов `getHeaderFieldKey()` и `getHeaderField()`. Кроме того, для удобства разработки предусмотрены перечисленные ниже методы обработки стандартных полей запроса.

```
getContentType()
getContentLength()
getContentEncoding()
getDate()
getExpiration()
getLastModified()
```

7. И наконец, для доступа к данным указанного ресурса следует вызвать метод `getInputStream()`, предоставляющий поток ввода для чтения данных. (Это тот же самый поток ввода, который возвращается методом `openStream()` из класса `URL`.) Существует также метод `getContent()`, но он не такой удобный. Для обработки содержимого стандартных типов, например, текста (`text/plain`) или изображений (`image/gif`), придется воспользоваться классами из пакета `com.sun`. Кроме того, можно зарегистрировать собственные обработчики содержимого, но они в данной книге не рассматриваются.

На заметку! Некоторые разработчики, пользующиеся классом `URLConnection`, ошибочно считают, что методы `getInputStream()` и `getOutputStream()` аналогичны одноименным методам из класса `Socket`. Это не совсем так. Класс `URLConnection` способен выполнять много других функций, в том числе обрабатывать заголовки запросов и ответов. Поэтому рекомендуется строго придерживаться указанной выше последовательности действий.

Рассмотрим методы из класса `URLConnection` более подробно. В нем имеется ряд методов, задающих свойства соединения еще до подключения к веб-серверу. Наиболее важными среди них являются методы `setDoInput()` и `setDoOutput()`. По умолчанию при соединении предоставляется поток ввода для чтения данных с веб-сервера, но не поток вывода для записи данных. Чтобы получить поток вывода (например, с целью разместить данные на веб-сервере), необходимо сделать следующий вызов:

```
connection.setDoOutput(true);
```

Далее можно установить ряд заголовков запроса и послать их веб-серверу в составе единого запроса. Ниже приведен пример заголовков запроса.

```
GET www.server.com/index.html HTTP/1.0
Referer: http://www.somewhere.com/links.html
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.4)
Host: www.server.com
Accept: text/html, image/gif, image/jpeg, image/png, /*
Accept-Language: en
Accept-Charset: iso-8859-1,* ,utf-8
Cookie: orangemilano=192218887821987
```

Метод `setIfModifiedSince()` служит для уведомления о том, что требуется получить только те данные, которые были изменены после определенной даты. Методы `setUseCaches()` и `setAllowUserInteraction()` следует вызывать только в аплетах. В частности, метод `setUseCaches()` предписывает браузеру проверить сначала кеш. А метод `setAllowUserInteraction()` позволяет открыть в аплете диалоговое окно с предложением ввести имя и пароль пользователя для защиты доступа к сетевым ресурсам (рис. 4.7).

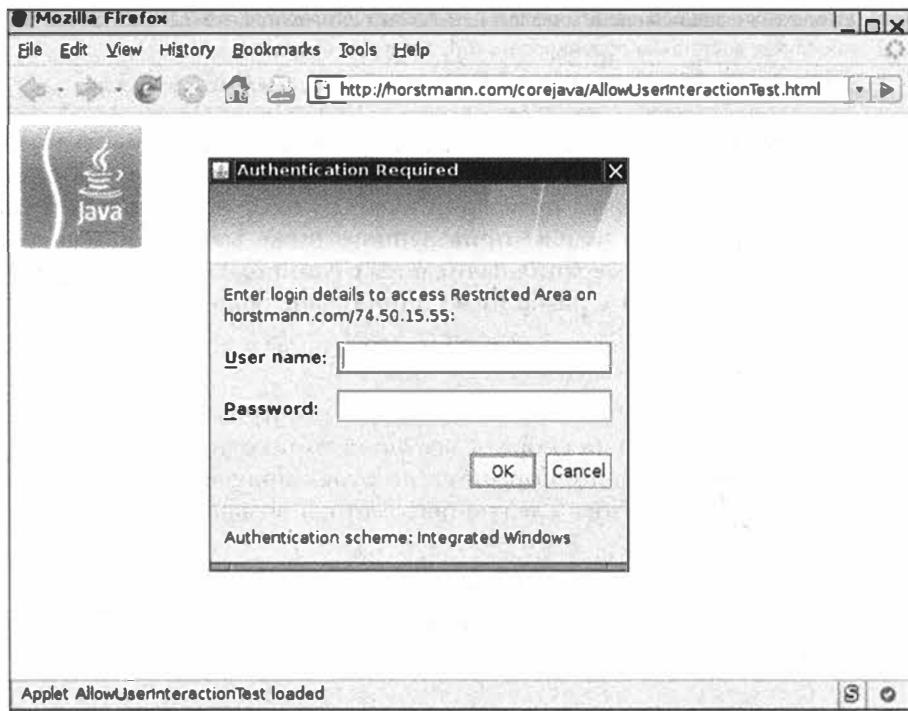


Рис. 4.7. Диалоговое окно для ввода имени и пароля доступа к сетевым ресурсам

И наконец, с помощью метода `setRequestProperty()` можно установить пару "имя–значение", имеющую определенный смысл для конкретного протокола. Формат заголовка запроса по сетевому протоколу HTTP описан в документе RFC 2616. Некоторые его параметры не очень хорошо документированы, поэтому за дополнительными разъяснениями зачастую приходится обращаться к другим программистам. Так, для доступа к защищенной паролем веб-странице необходимо выполнить следующие действия.

1. Составить символьную строку из имени пользователя, двоеточия и пароля:
`String input = username + ":" + password;`
2. Перекодировать полученную в итоге символьную строку по алгоритму кодирования Base64, как показано ниже. (Этот алгоритм преобразует последовательность байтов в последовательность символов в коде ASCII.)

```
Base64.Encoder encoder = Base64.getEncoder();
String encoding =
    encoder.encodeToString(input.getBytes(StandardCharsets.UTF_8));
```

3. Вызвать метод `setRequestProperty()` с именем свойства "Authorization" и значением "Basic " + `encoding`:

```
connection.setRequestProperty("Authorization", "Basic " + encoding);
```



Совет! Здесь рассматривается способ обращения к защищенной паролем веб-странице. А для доступа к защищенному паролем FTP-файлу применяется совершенно другой подход. В этом случае достаточно сформировать URL следующего вида:

ftp://имя_пользователя:пароль@ftp.ваш_сервер.com/pub/file.txt

После вызова метода `connect()` можно запросить данные заголовка из ответа. Рассмотрим сначала способ перечисления всех полей заголовка. Создатели рассматриваемого здесь класса посчитали нужным создать собственный способ перебора полей. Так, при вызове приведенного ниже метода получается *n*-й ключ заголовка, причем нумерация начинается с единицы! В итоге возвращается пустое значение `null`, если *n* равно нулю или больше общего количества полей заголовка.

```
String key = connection.getHeaderFieldKey(n);
```

Но для определения количества полей не предусмотрено никакого другого метода. Чтобы перебрать все поля, приходится вызывать метод `getHeaderFieldKey()` до тех пор, пока не будет получено пустое значение `null`. Аналогично при вызове следующего метода возвращается значение из *n*-го поля:

```
String value = connection.getHeaderField(n);
```

Метод `getHeaderFields()` возвращает объект типа `Map` с полями заголовка, как показано ниже.

```
Map<String, List<String>> headerFields = connection.getHeaderFields();
```

В качестве примера ниже приведен ряд полей заголовка из типичного ответа на запрос по сетевому протоколу HTTP.

```
Date: Wed, 27 Aug 2008 00:15:48 GMT
Server: Apache/2.2.2 (Unix)
Last-Modified: Sun, 22 Jun 2008 20:53:38 GMT
Accept-Ranges: bytes
Content-Length: 4813
Connection: close
Content-Type: text/html
```

Для удобства разработки предусмотрены шесть методов, получающих значения из наиболее употребительных полей заголовка и приводящие эти значения к соответствующим числовым типам по мере необходимости. Все эти удобные методы перечислены в табл. 4.1. В методах, возвращающих значения типа `long`, отсчет количества возвращаемых секунд начинается с полуночи 1 января 1970 г.

Таблица 4.1. Удобные методы, получающие значения полей заголовка из ответа на запрос

Имя поля (ключа)	Имя метода	Возвращаемое значение
Date	getDate	long
Expires	getExpiration	long
Last-Modified	getLastModified	long
Content-Length	getContentLength	int
Content-Type	getContentType	String
Content-Encoding	getContentEncoding	String

В примере программы, исходный код которой приведен в листинге 4.6, предоставляется возможность поэкспериментировать с соединениями по URL. Запустив программу, вы можете указать в командной строке конкретный URL, имя пользователя и пароль:

```
java urlConnection.URLConnectionTest http://www.ваш_сервер.com
пользователь пароль
```

В итоге программа выведет на экран следующее.

- Все ключи и значения из полей заголовка.
- Значения, возвращаемые шестью служебными методами доступа к наиболее употребительным полям заголовка (табл. 4.1).
- Первые 10 символьных строк из запрашиваемого ресурса.

Листинг 4.6. Исходный код из файла urlConnection/URLConnectionTest.java

```

1 package urlConnection;
2
3 import java.io.*;
4 import java.net.*;
5 import java.nio.charset.*;
6 import java.util.*;
7
8 /**
9  * В этой программе устанавливается соединение по заданному URL и
10 * отображаются данные заголовка из получаемого ответа, а также
11 * первые 10 строк запрашиваемых данных. Для этого в командной
12 * строке следует указать конкретный URL и дополнительно имя
13 * пользователя и пароль (для элементарной аутентификации по
14 * сетевому протоколу HTTP)
15 * @version 1.11 2007-06-26
16 * @author Cay Horstmann
17 */
18 public class URLConnectionTest
19 {
20     public static void main(String[] args)
21     {
22         try
23         {
24             String urlName;
25             if (args.length > 0) urlName = args[0];
26             else urlName = "http://horstmann.com";
27
28             URL url = new URL(urlName);

```

```
29     URLConnection connection = url.openConnection();
30
31     // установить имя пользователя и пароль, если они
32     // указаны в командной строке
33
34     if (args.length > 2)
35     {
36         String username = args[1];
37         String password = args[2];
38         String input = username + ":" + password;
39         Base64.Encoder encoder = Base64.getEncoder();
40         String encoding = encoder.encodeToString(
41             input.getBytes(StandardCharsets.UTF_8));
42         connection.setRequestProperty("Authorization",
43             "Basic " + encoding);
43     }
44
45     connection.connect();
46
47     // вывести поля заголовка
48
49     Map<String, List<String>> headers =
50         connection.getHeaderFields();
51     for (Map.Entry<String,
52             List<String>"> entry : headers.entrySet())
53     {
54         String key = entry.getKey();
55         for (String value : entry.getValue())
56             System.out.println(key + ": " + value);
57     }
58
59     // вывести значения полей заголовка,
60     // используя удобные методы
61
62     System.out.println("-----");
63     System.out.println("getContentType: "
64         + connection.getContentType());
65     System.out.println("getContentLength: "
66         + connection.getContentLength());
67     System.out.println("getContentEncoding: "
68         + connection.getContentEncoding());
69     System.out.println("getDate: " + connection.getDate());
70     System.out.println("getExpiration: "
71         + connection.getExpiration());
72     System.out.println("getLastModified: "
73         + connection.getLastModified());
74     System.out.println("-----");
75
76     String encoding = connection.getContentEncoding();
77     if (encoding == null) encoding = "UTF-8";
78     try (Scanner in = new Scanner(connection.getInputStream(),
79                                     encoding))
80     {
81         // вывести первые десять строк запрашиваемого содержимого
82
83         for (int n = 1; in.hasNextLine() && n <= 10; n++)
84             System.out.println(in.nextLine());
85         if (in.hasNextLine()) System.out.println("... ");
86     }
87 }
```

```
89     catch (IOException e)
90     {
91         e.printStackTrace();
92     }
93 }
94 }
```

java.net.URL 1.0

- **InputStream openStream()**

Открывает поток ввода для чтения данных из ресурса.

- **URLConnection openConnection()**

Возвращает объект типа **URLConnection**, управляющий соединением с ресурсом.

java.netURLConnection 1.0

- **void setDoInput(boolean doInput)**

- **boolean getDoInput()**

Если параметр **doInput** принимает логическое значение **true**, пользователь может принимать вводимые данные из текущего объекта типа **URLConnection**.

- **void setDoOutput(boolean doOutput)**

- **boolean getDoOutput()**

Если параметр **doOutput** принимает логическое значение **true**, пользователь может передавать выводимые данные в текущий объект типа **URLConnection**.

- **void setIfModifiedSince(long time)**

- **long getIfModifiedSince()**

Свойство **ifModifiedSince** настраивает данный объект типа **URLConnection** на извлечение только тех данных, которые были изменены после указанного момента времени. Время задается в секундах, начиная с полуночи 1 января 1970 г. по Гринвичу.

- **void setUseCaches(boolean useCaches)**

- **boolean getUseCaches()**

Если параметр **useCaches** принимает логическое значение **true**, данные можно извлечь из локального кеша. Однако кеш не поддерживается самим объектом типа **URLConnection**, поэтому он должен быть предоставлен внешней программой, например, браузером.

- **void setAllowUserInteraction(boolean allowUserInteraction)**

- **boolean getAllowUserInteraction()**

Если параметр **allowUserInteraction** принимает логическое значение, у пользователя может запрашиваться пароль. Следует, однако, иметь в виду, что у самого объекта типа **URLConnection** отсутствуют средства, требующиеся для выполнения подобных запросов. Поэтому запрос пароля должен быть организован во внешней программе, например, в браузере или подключаемом модуле.

- **void setConnectTimeout(int timeout) 5.0**

- **int getConnectTimeout() 5.0**

Устанавливают или возвращают величину времени ожидания (в миллисекундах) для соединения. Если время ожидания истечет до установления соединения, метод **connect()** из соответствующего потока ввода сгенерирует исключение типа **SocketTimeoutException**.

java.net.URLConnection 1.0 (окончание)

- **void setReadTimeout(int timeout) 5.0**
- **int getReadTimeout() 5.0**

Устанавливают или возвращают величину времени ожидания [в миллисекундах] для чтения данных. Если время ожидания истечет до успешного завершения операции чтения, метод **read()** генерирует исключение типа **SocketTimeoutException**.

- **void setRequestProperty(String key, String value)**

Устанавливает значение в поле заголовка.

- **Map<String, List<String>> getRequestProperties() 1.4**

Возвращает отображение со свойствами запроса. Все свойства по одному и тому же ключу вносятся в список.

- **void connect()**

Устанавливает соединение с удаленным ресурсом и получает данные заголовка из ответа.

- **Map<String, List<String>> getHeaderFields() 1.4**

Возвращает отображение с полями заголовка из ответа. Все свойства одного и того же ключа вносятся в список.

- **String getHeaderFieldKey(int n)**

Возвращает ключ **n**-го поля заголовка из ответа или пустое значение **null**, если **n** меньше или равно нулю или превышает количество полей.

- **String getHeaderField(int n)**

Возвращает значение **n**-го поля заголовка из ответа или пустое значение **null**, если **n** меньше или равно нулю или превышает количество полей.

- **int getContentLength()**

Возвращает длину доступного содержимого или **-1**, если длина неизвестна.

- **String getContentType()**

Возвращает тип содержимого, например **text/plain** или **image/gif**.

- **String getContentEncoding()**

Возвращает кодировку содержимого, например **gzip**. Применяется редко, потому что используемая по умолчанию кодировка не всегда указывается в поле **identity** заголовка **Content-Encoding**.

- **long getDate()**

- **long getExpiration()**

- **long getLastModified()**

Возвращают время создания, последней модификации ресурса или время, когда срок действия ресурса истекает. Время указывается в секундах, начиная с 1 января 1970 г. по Гринвичу.

- **InputStream getInputStream()**

- **OutputStream getOutputStream()**

Возвращают поток ввода для чтения данных из ресурса или вывода для записи данных в ресурс.

- **Object getContent()**

Выбирает подходящий обработчик содержимого для чтения данных из ресурса. Этот метод вряд ли полезен для чтения данных стандартного типа, например, **text/plain** или **image/gif**, кроме тех случаев, когда требуется создать собственный обработчик этих типов данных.

4.4.3. Отправка данных формы

В предыдущем разделе описывался способ чтения данных с веб-сервера, а в этом разделе рассматривается способ передачи данных из клиентской программы на веб-сервер, а также другим программам, которые может вызывать веб-сервер. Для передачи данных из браузера на веб-сервер нужно заполнить форму, аналогичную приведенной на рис. 4.8.

Рис. 4.8. HTML-форма

Когда пользователь щелкает на кнопке **Submit** (Отправить), данные, введенные в текстовых полях, а также сведения о состоянии флагов и кнопок-переключателей передаются на веб-сервер. Получив данные, введенные пользователем в форме, веб-сервер вызывает программу для их последующей обработки.

Существует целый ряд технологий, позволяющих веб-серверу вызывать программы для обработки данных. Наиболее часто для этой цели используются сервлеты на Java, JavaServer Faces, Microsoft ASP (Active Server Pages) и сценарии CGI.

Программа, выполняющаяся на стороне сервера, обрабатывает данные, введенные пользователем в форме, и формирует новую HTML-страницу, которую веб-сервер передает обратно браузеру. Последовательность действий

по обработке данных из формы условно показана на рис. 4.9. Страница, сформированная сервером, может содержать новые данные (например, результаты поиска) или только подтверждение о получении введенных данных. Здесь и далее не рассматриваются вопросы реализации серверных программ, а основное внимание уделяется написанию клиентских программ, предназначенных для взаимодействия с готовыми сценариями.

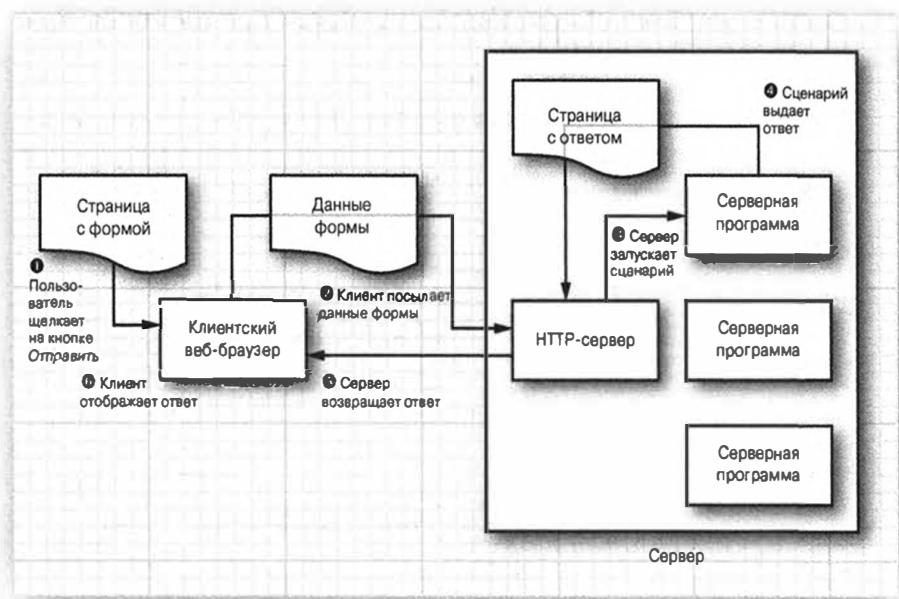


Рис. 4.9. Порядок обработки данных в серверной программе

При передаче данных на веб-сервер не имеет никакого значения, будет ли использован для их интерпретации сценарий CGI, сервлет или программа другого типа. Клиент посыпает данные на веб-сервер в стандартном формате, а веб-сервер должен сам найти ту программу, которая выдаст нужный ответ.

Передача данных на веб-сервер может осуществляться по командам GET и POST. При выдаче команды GET параметры просто указываются в конце URL в следующем формате:

`http://хост/сценарий?параметры`

Каждый параметр имеет вид `имя=значение`. Параметры разделяются знаками &. Значения параметров кодируются по схеме кодирования URL, которая подчиняется следующим правилам.

- Символы от A до Z, от a до z, от 0 до 9, а также знаки ., -, ~ и _ остаются без изменения.
- Все пробелы заменяются знаками +.
- Все остальные символы кодируются в кодировке UTF-8, а каждый байт преобразуется к виду %UV, где UV — двухзначное шестнадцатеричное число.

Например, название города и штата *San Francisco, CA* передается в закодированном виде как *San+Francisco%2c+CA*. Здесь шестнадцатеричное число *2c* (или десятичное *44*) обозначает запятую в кодировке UTF-8. Благодаря такому способу кодирования промежуточные программы не будут путаться в пробелах и смогут правильно интерпретировать другие символы.

Так, на момент написания данной книги веб-сайт Google Maps (www.google.com/maps) принимал параметры запроса с именами *q* и *hl*, значения которых определяют местоположение и естественный язык в ответе. Чтобы получить карту местности по адресу Маркет-стрит, 1, г. Сан-Франциско на немецком языке, необходимо указать следующий URL:

```
http://www.google.com/maps?q=1+Market+Street+San+Francisco&hl=de
```

Очень длинные строки запроса могут выглядеть непривлекательно в большинстве браузеров, а в старых браузерах и промежуточных серверах накладывается ограничение на количество символов, включаемых в запрос по команде GET. Именно поэтому запрос по команде POST чаще всего употребляется для форм, содержащих немало данных. Параметры запроса по команде POST не следует включать в состав URL. Вместо этого нужно получить поток вывода из объекта типа *URLConnection* и записать в него пары "имя–значение". Кроме того, значения, включаемые в URL, необходимо закодировать, разделив их знаком **&**.

Рассмотрим этот процесс более подробно. Для передачи данных серверной программе сначала создается объект типа *URLConnection* следующим образом:

```
URL url = new URL("http://хост/сценарий");
URLConnection connection = url.openConnection();
```

Затем вызывается метод *setDoOutput()*, чтобы установить соединение для передачи данных:

```
connection.setDoOutput(true);
```

Далее вызывается метод *getOutputStream()*, чтобы получить поток вывода. Для передачи текстовых данных поток вывода удобно инкапсулировать в объект типа *PrintWriter* следующим образом:

```
PrintWriter out =
    new PrintWriter(connection.getOutputStream(), "UTF-8");
```

Теперь можно передать данные на сервер, как следует из приведенного ниже фрагмента кода.

```
out.print(name1 + "=" + URLEncoder.encode(value1, "UTF-8") + "&");
out.print(name2 + "=" + URLEncoder.encode(value2, "UTF-8"));
```

После передачи данных выходной поток вывода закрывается следующим образом:

```
out.close();
```

И, наконец, вызывается метод *getInputStream()*, чтобы прочитать ответ с сервера.

Рассмотрим конкретный практический пример. Веб-сайт под адресу <https://www.usps.com/zip4> содержит страницу с формой для поиска почтового индекса по введенному адресу улицы (см. рис. 4.8). Чтобы воспользоваться этой формой в программе на Java, нужно знать URL и параметры запроса по команде POST.

Эту информацию можно было бы получить, просмотрев код HTML-разметки формы, но ее, как правило, проще “выудить” из запроса с помощью сетевого монитора, входящего в набор инструментальных средств веб-разработки в большинстве браузеров. В качестве примера на рис. 4.10 приведен моментальный снимок, сделанный сетевым монитором браузера Firefox при передаче на обработку данных выбранному для данного примера веб-сайту. На этом моментальном снимке можно выявить URL, параметры и значения, указанные при передаче данных на обработку.



Рис. 4.10. HTML-форма

При передаче данных формы на обработку в HTTP-заголовок включается тип содержимого и его длина, как показано ниже.

Content-Type: application/x-www-form-urlencoded

Данные можно передать и в других форматах. Так, если данные посылаются в формате JSON (JavaScript Object Notation — Представление объектов JavaScript), в HTTP-заголовке запроса по команде POST должен быть указан тип содержимого application/json, а также его длина, как показано в следующем примере:

Content-Length: 124

В листинге 4.7 приведен исходный код примера программы, посылающей данные на сервер по команде POST. В файл свойств с расширением .properties вводятся следующие данные:

```
url=https://tools.usps.com/go/ZipLookupAction.action
tAddress=1 Market Street
tCity=San Francisco
sState=CA
...
```

Программа удаляет элемент url, а все остальные элементы направляет методу doPost(). В методе doPost() сначала устанавливается соединение, затем вызывается метод setDoOutput(true) и открывается поток вывода. Затем перечисляются все ключи и значения. Для каждой пары "ключ-значение" по очереди передаются ключ, знак =, значение и разделительный знак &:

```
out.print(key);
out.print('=');
out.print(URLEncoder.encode(value, "UTF-8"));
if (дополнительные пары "ключ-значение") out.print('&');
```

Взаимодействие с сервером фактически происходит при переходе от записи к чтению любой части ответа. В заголовке Content-Length задается длина выводимых данных, а в заголовке Content-Type — тип application/x-www-form-urlencoded, если только не был указан другой тип содержимого. Заголовки и данные запроса посылаются серверу. Затем читаются заголовки и данные ответа сервера, которые могут быть запрошены. В данном примере программы такой переход от записи к чтению происходит при вызове метода connection.getContentEncoding().

Следует, однако, иметь в виду, что если при выполнении серверной программы возникнет ошибка, то вызов метода connection.getInputStream() приведет к исключению типа FileNotFoundException. Тем не менее сервер продолжит передачу данных, отправив HTML-страницу с сообщением об ошибке. Обычно это сообщение "Error 404-page not found", уведомляющее о том, что данная страница не найдена. Для перехвата этой страницы следует вызвать метод getErrorStream():

```
InputStream err = connection.getErrorStream();
```



На заметку! Метод `getErrorStream()`, а также ряд других методов, применяемых в рассматриваемом здесь примере программы, относятся к классу `HttpURLConnection`, производному от класса `URLConnection`. Если сделать запрос по URL, начинающемуся с префикса `http://` или `https://`, то полученный в итоге объект соединения можно привести к типу `HttpURLConnection`.

При передаче данных по команде POST на сервер серверная программа может *переадресовать* его по другому URL для получения искомой информации. Сервер может сделать это потому, что искомая информация находится в каком-нибудь другом месте. А с другой стороны, он может предоставить отмеченный закладкой URL. Как правило, класс HttpURLConnection может осуществить переадресацию.



На заметку! Если при переадресации требуется переслать cookie-файлы из одного сайта на другой, с этой целью можно настроить глобальный обработчик cookie-файлов следующим образом:

```
CookieHandler.setDefault(new CookieManager(null,  
                                         CookiePolicy.ACCEPT_ALL));
```

В этом случае cookie-файлы будут надлежащим образом включены в переадресацию.

Несмотря на то что переадресация, как правило, осуществляется автоматически, иногда это приходится делать вручную. Автоматическая переадресация между сетевыми протоколами HTTP и HTTPS не поддерживается из соображений безопасности. Она может не состояться и по менее понятным причинам. Например, упомянутая выше веб-служба определения почтовых индексов не действует, если параметр запроса User-Agent содержит строку Java. Такое поведение можно объяснить тем, что почтовая служба не желает обслуживать программные запросы. И хотя в первоначальном запросе можно задать другую строку для пользовательского посредника, такая настройка не используется при автоматической переадресации, при которой всегда посылается типичная строка пользовательского посредника, содержащая слово Java.

В подобных случаях переадресацию можно осуществить вручную. Прежде чем подключиться к серверу, необходимо выключить режим автоматической переадресации следующим образом:

```
connection.setInstanceFollowRedirects(false);
```

Сделав запрос, следует получить код ответа:

```
int responseCode = connection.getResponseCode();
```

и проверить, относится ли он к одному из перечисленных ниже кодов.

```
HttpURLConnection.HTTP_MOVED_PERM  
HttpURLConnection.HTTP_MOVED_TEMP  
HttpURLConnection.HTTP_SEE_OTHER
```

В таком случае следует сначала получить заголовок ответа Location, а затем URL для переадресации. Далее необходимо разорвать текущее соединение и установить другое соединение по новому URL:

```
String location = connection.getHeaderField("Location");  
if (location != null)  
{  
    URL base = connection.getURL();  
    connection.disconnect();  
    connection = (HttpURLConnection)  
        new URL(base, location).openConnection();  
    . . .  
}
```

Приемы, демонстрируемые в рассматриваемой здесь программе, могут оказаться полезными всякий раз, когда требуется запросить информацию из существующего веб-сайта. Для этого достаточно выяснить сначала параметры, которые требуется послать в запросе, а затем удалить дескрипторы HTML-разметки и прочую ненужную информацию из полученного ответа.



На заметку! Как видите, классами из библиотеки Java вполне можно пользоваться для взаимодействия с веб-страницами, хотя это и не совсем удобно. Для этой цели лучше все же воспользоваться библиотекой Apache HttpClient (<http://hc.apache.org/httpcomponents-client-ga>).

Листинг 4.7. Исходный код из файла post/PostTest.java

```
1 package post;
2
3 import java.io.*;
4 import java.net.*;
5 import java.nio.file.*;
6 import java.util.*;
7
8 /**
9  * В этой программе демонстрируется применение класса
10 * URLConnection для формирования запроса по команде POST
11 * @version 1.40 2016-04-24
12 * @author Cay Horstmann
13 */
14 public class PostTest
15 {
16     public static void main(String[] args) throws IOException
17     {
18         String propsFilename =
19             args.length > 0 ? args[0] : "post/post.properties";
20         Properties props = new Properties();
21         try (InputStream in =
22             Files.newInputStream(Paths.get(propsFilename)))
23         {
24             props.load(in);
25         }
26         String urlString = props.remove("url").toString();
27         Object userAgent = props.remove("User-Agent");
28         Object redirects = props.remove("redirects");
29         CookieHandler.setDefault(new CookieManager(null,
30                                         CookiePolicy.ACCEPT_ALL));
31         String result = doPost(new URL(urlString), props,
32             userAgent == null ? null : userAgent.toString(),
33             redirects == null ? -1 :
34                 Integer.parseInt(redirects.toString()));
35         System.out.println(result);
36     }
37
38 /**
39  * Сделать HTTP-запрос по команде POST
40  * @param url Конкретный URL для отправки запроса
41  * @param nameValuePairs Параметры запроса
42  * @param userAgent Пользовательский посредник или пустое
43  * значение null, если используется посредник по умолчанию
```

```
44     * @param redirects Количество последующих переадресаций
45     * вручную или значение -1, если переадресация автоматическая
46     * @return Данные, возвращаемые из сервера
47 */
48 public static String doPost(URL url, Map<Object,
49                             Object> nameValuePairs, String userAgent, int redirects)
50         throws IOException
51 {
52     HttpURLConnection connection =
53         (HttpURLConnection) url.openConnection();
54     if (userAgent != null)
55         connection.setRequestProperty("User-Agent", userAgent);
56
57     if (redirects >= 0)
58         connection.setInstanceFollowRedirects(false);
59
60     connection.setDoOutput(true);
61
62     try (PrintWriter out =
63          new PrintWriter(connection.getOutputStream()))
64     {
65         boolean first = true;
66         for (Map.Entry<Object, Object> pair :
67              nameValuePairs.entrySet())
68         {
69             if (first) first = false;
70             else out.print('&');
71             String name = pair.getKey().toString();
72             String value = pair.getValue().toString();
73             out.print(name);
74             out.print('=');
75             out.print(URLEncoder.encode(value, "UTF-8"));
76         }
77     }
78     String encoding = connection.getContentEncoding();
79     if (encoding == null) encoding = "UTF-8";
80
81     if (redirects > 0)
82     {
83         int responseCode = connection.getResponseCode();
84         if (responseCode == HttpURLConnection.HTTP_MOVED_PERM
85             || responseCode == HttpURLConnection.HTTP_MOVED_TEMP
86             || responseCode == HttpURLConnection.HTTP_SEE_OTHER)
87         {
88             String location = connection.getHeaderField("Location");
89             if (location != null)
90             {
91                 URL base = connection.getURL();
92                 connection.disconnect();
93                 return doPost(new URL(base, location), nameValuePairs,
94                               userAgent, redirects - 1);
95             }
96         }
97     }
98 }
99 else if (redirects == 0)
100 {
101     throw new IOException("Too many redirects");
102 }
```

```

104     StringBuilder response = new StringBuilder();
105     try (Scanner in = new Scanner(connection.getInputStream(),
106                                     encoding))
107     {
108         while (in.hasNextLine())
109         {
110             response.append(in.nextLine());
111             response.append("\n");
112         }
113     }
114     catch (IOException e)
115     {
116         InputStream err = connection.getErrorStream();
117         if (err == null) throw e;
118         try (Scanner in = new Scanner(err))
119         {
120             response.append(in.nextLine());
121             response.append("\n");
122         }
123     }
124
125     return response.toString();
126 }
127 }
```

java.net.HttpURLConnection 1.0

- **InputStream getErrorStream()**

Возвращает поток ввода, из которого читаются сообщения сервера об ошибках.

java.net.URLEncoder 1.0

- **static String encode(String s, String encoding) 1.4**

Возвращает строку *s*, кодированную в формате URL с использованием заданной кодировки символов. (Рекомендуется указывать кодировку "UTF-8".) При кодировании в формате URL символы 'A'-'Z', 'a'-'z', '0'-'9', '-' , '_' , '.' и '~' оставляются без изменения. Пробелы заменяются знаками '+', а все остальные символы — последовательностями кодированных байтов в форме "%XY", где **0xXY** — шестнадцатеричное значение байта.

java.net.URLDecoder 1.2

- **static String decode(String s, String encoding) 1.4**

Возвращает декодированную форму строки *s*, кодированной в формате URL, с использованием заданной кодировки символов.

4.5. Отправка электронной почты

В прошлом для отправки электронной почты достаточно было написать программу, устанавливавшую соединение с сокетом через порт 25, который обычно используется для работы сетевого протокола SMTP (Simple Mail Transport

Protocol — простой протокол передачи почты), описывающего формат электронных сообщений. После подключения к серверу в данной программе нужно было послать заголовок сообщения, который достаточно просто создать в формате SMTP, а затем и текст сообщения, выполнив перечисленные ниже действия.

1. Открыть сокет на своем компьютере, подключенном к Интернету, как показано ниже.

```
Socket s = new Socket("mail.yourserver.com", 25); // номер порта 25
                                                    // соответствует протоколу SMTP
PrintWriter out = new PrintWriter(s.getOutputStream(), "UTF-8");
```

2. Направить в поток вывода следующие данные:

```
HELO компьютер отправителя
MAIL FROM: адрес отправителя
RCPT TO: адрес получателя
DATA
почтовое сообщение
(любое количество строк)

QUIT
```

В спецификации протокола SMTP (документ RFC 821) требуется, чтобы строчки завершались комбинациями символов /r и /n. Первоначально SMTP-серверы исправно направляли электронную почту от любого адресата. Но когда навязчивые сообщения наводнили Интернет, большинство этих серверов были оснащены встроенными проверками, принимая запросы только по тем IP-адресам, которым они доверяют. Аутентификация обычно происходит через безопасные сокетные соединения.

Реализовать алгоритмы подобной аутентификации вручную — дело непростое. Поэтому в этом разделе будет показано, как пользоваться прикладным программным интерфейсом JavaMail API для отправки сообщений электронной почты из программы на Java. С этой целью загрузите данный прикладной интерфейс по адресу www.oracle.com/technetwork/java/javamail и разархивируйте его на жесткий диск своего компьютера.

Чтобы воспользоваться прикладным интерфейсом JavaMail API, необходимо установить некоторые свойства, зависящие от конкретного почтового сервера. В качестве примера ниже приведены свойства, устанавливаемые для почтового сервера GMail. Оничитываются из файла свойств в рассматриваемом здесь примере программы, исходный код которой приведен в листинге 4.8.

```
mail.transport.protocol=smtpls
mail.smtps.auth=true
mail.smtps.host=smtp.gmail.com
mail.smtps.user=cayhorstmann@gmail.com
```

Из соображений безопасности пароль не вводится в файл свойств, но предлагается для ввода вручную. После чтения из файла свойств сеанс почтовой связи устанавливается следующим образом:

```
Session mailSession = Session.getDefaultInstance(props);
```

Затем составляется почтовое сообщение с указанием требуемого отправителя, получателя, темы и текста самого сообщения, как показано ниже.

```
MimeMessage message = new MimeMessage(mailSession);
message.setFrom(new InternetAddress(from));
message.addRecipient(RecipientType.TO, new InternetAddress(to));
message.setSubject(subject);
message.setText(builder.toString());
```

Далее почтовое сообщение отправляется следующим образом:

```
Transport tr = mailSession.getTransport();
tr.connect(null, password);
tr.sendMessage(message, message.getAllRecipients());
tr.close();
```

Рассматриваемая здесь программа читает почтовое сообщение из текстового файла в приведенном ниже формате.

Отправитель *Получатель* *Тема* *Текст сообщения* (любое количество строк)

Для запуска данной программы на выполнение введите приведенную ниже команду, где `mail.jar` — архивный JAR-файл, входящий в состав распространяемой версии прикладного программного интерфейса JavaMail API. (При указании пути к классу пользователи Windows должны ввести после параметра `-classpath` точку с запятой вместо двоеточия.)

```
java -classpath .:path/to/mail.jar path/to/message.txt
```

На момент написания данной книги почтовый сервер GMail не проверял достоверность получаемой информации, а следовательно, в почтовом сообщении можно было указать любого отправителя. (Это обстоятельство следует иметь в виду при получении от отправителя по адресу `president@whitehouse.gov` очередного приглашения на официальный прием, организуемый на лужайке перед Белым домом.)



Совет! Если вам не удастся выяснить причину, по которой соединение с почтовым сервером не действует, сделайте следующий вызов и проверьте почтовые сообщения:

```
mailSession.setDebug(true);
```

Кроме того, обратитесь за полезными советами на веб-страницу **JavaMail API FAQ** (Часто задаваемые вопросы по прикладному программному интерфейсу JavaMail API FAQ) по адресу <http://www.oracle.com/technetwork/java/javamail/faq-135477.html>.

Листинг 4.8. Исходный код из файла mail/MailTest.java

```
1 package mail;
2
3 import java.io.*;
4 import java.nio.charset.*;
5 import java.nio.file.*;
6 import java.util.*;
7 import javax.mail.*;
8 import javax.mail.internet.*;
9 import javax.mail.internet.MimeMessage.RecipientType;
10
11 /**
12 * В этой программе демонстрируется применение прикладного
13 * программного интерфейса JavaMail API для отправки сообщений
14 * по электронной почте
```

```
15 * @author Cay Horstmann
16 * @version 1.00 2012-06-04
17 */
18 public class MailTest
19 {
20     public static void main(String[] args)
21         throws MessagingException, IOException
22     {
23         Properties props = new Properties();
24         try (InputStream in = Files.newInputStream(Paths.get(
25             "mail", "mail.properties")))
26         {
27             props.load(in);
28         }
29         List<String> lines = Files.readAllLines(
30             Paths.get(args[0]), Charset.forName("UTF-8"));
31
32         String from = lines.get(0);
33         String to = lines.get(1);
34         String subject = lines.get(2);
35
36         StringBuilder builder = new StringBuilder();
37         for (int i = 3; i < lines.size(); i++)
38         {
39             builder.append(lines.get(i));
40             builder.append("\n");
41         }
42
43         Console console = System.console();
44         String password = new String(
45             console.readPassword("Password: "));
46
47         Session mailSession = Session.getDefaultInstance(props);
48         // mailSession.setDebug(true);
49         MimeMessage message = new MimeMessage(mailSession);
50         message.setFrom(new InternetAddress(from));
51         message.addRecipient(RecipientType.TO,
52             new InternetAddress(to));
53         message.setSubject(subject);
54         message.setText(builder.toString());
55         Transport tr = mailSession.getTransport();
56         try
57         {
58             tr.connect(null, password);
59             tr.sendMessage(message, message.getAllRecipients());
60         }
61         finally
62         {
63             tr.close();
64         }
65     }
66 }
```

В этой главе было показано, каким образом на Java пишется исходный код программ для сетевых клиентов и серверов и как организуется сбор данных с веб-серверов. А в следующей главе речь пойдет о взаимодействии с базами данных. Из нее вы узнаете, как работать с реляционными базами данных в программах на Java, используя прикладной программный интерфейс JDBC API.

ГЛАВА

5

Работа с базами данных

В этой главе...

- ▶ Структура JDBC
- ▶ Язык SQL
- ▶ Конфигурирование JDBC
- ▶ Работа с операторами JDBC
- ▶ Выполнение запросов
- ▶ Прокручиваемые и обновляемые результирующие наборы
- ▶ Наборы строк
- ▶ Метаданные
- ▶ Транзакции
- ▶ Расширенные типы данных SQL
- ▶ Управление подключением к базам данных в веб-приложениях и производственных приложениях

В 1996 году компания Sun Microsystems выпустила первую версию прикладного программного интерфейса API для организации доступа из программ на Java к базам данных (JDBC). Этот прикладной программный интерфейс позволяет соединяться с базой данных, запрашивать и обновлять данные с помощью языка структурированных запросов (Structured Query Language – SQL). Язык SQL фактически стал стандартным средством взаимодействия с реляционными базами данных. С тех пор JDBC стал одним из наиболее употребительных прикладных программных интерфейсов API в библиотеке Java.

Средства JDBC неоднократно обновлялись. В состав комплекта JDK 1.2, выпущенного в 1998 году, была включена версия JDBC 2. А версия JDBC 3 была включена в версии Java SE 1.4 и 5.0. На момент написания данной книги последней была версия JDBC 4.2, вошедшая в состав Java SE 8.

В этой главе рассматриваются принципы, положенные в основу прикладного интерфейса JDBC. Из нее вы узнаете (а возможно, лишь вспомните) о языке SQL, который является стандартным средством доступа к реляционным базам данных. В ней будут также рассмотрены примеры применения интерфейса JDBC, демонстрирующие наиболее распространенные приемы обращения с базами данных в прикладных программах.



На заметку! Как заявляют в компании Oracle, JDBC — это торговая марка, а не сокращение Java Database Connectivity. Она была придумана по аналогии с обозначением ODBC стандартного прикладного интерфейса для работы с базами данных, который был первоначально предложен корпорацией Microsoft и затем внедрен в стандарт SQL.

5.1. Структура JDBC

Создатели Java с самого начала осознавали потенциальные преимущества данного языка для работы с базами данных. С 1995 года они начали работать над расширением стандартной библиотеки Java для организации доступа к базам данных средствами SQL. Сначала они попробовали создать такие расширения Java, которые позволили бы осуществлять доступ к произвольной базе данных только средствами Java, но очень скоро убедились в бесперспективности такого подхода, поскольку для доступа к базам данных применялись самые разные протоколы. Кроме того, поставщики программного обеспечения баз данных были весьма заинтересованы в разработке на Java стандартного сетевого протокола для доступа к базам данных, но при условии, что за основу будет принят их *собственный* сетевой протокол.

В конечном счете поставщики баз данных и инструментальных средств для доступа к ним сошлись на том, что лучше предоставить прикладной программный интерфейс API только на Java для доступа к базам данных средствами SQL, а также диспетчер драйверов, который позволил бы подключать к базам драйверы независимых производителей. Такой подход позволял поставщикам баз данных создавать собственные драйверы, которые подключались бы с помощью данного диспетчера. Предполагалось, что это будет простой механизм регистрации драйверов.

Подобная организация прикладного интерфейса JDBC основана на весьма удачной модели интерфейса ODBC, разработанного в корпорации Microsoft. В основу интерфейсов JDBC и ODBC положен общий принцип: программы, написанные в соответствии с требованиями прикладного программного интерфейса API, способны взаимодействовать с диспетчером драйверов JDBC, который, в свою очередь, использует подключаемые драйверы для обращения к базе данных. Это означает, что для работы с базами данных в прикладных программах достаточно пользоваться средствами JDBC API.

5.1.1. Типы драйверов JDBC

Каждый драйвер JDBC принадлежит к одному из перечисленных ниже типов.

- **Драйвер типа 1.** Преобразует интерфейс JDBC в ODBC и для взаимодействия с базой данных использует драйвер ODBC. Один такой драйвер был

включен в первые версии Java под названием *мост JDBC/ODBC*. Но для его применения требуется установить и настроить соответствующим образом драйвер ODBC. В первом выпуске JDBC этот мост предполагалось использовать только для тестирования, а не для применения в рабочих программах. В настоящее время уже имеется достаточное количество более удачных драйверов, поэтому пользоваться мостом JDBC/ODBC не рекомендуется.

- **Драйвер типа 2.** Разрабатывается преимущественно на Java и частично на собственном языке программирования, который используется для взаимодействия с клиентским прикладным программным интерфейсом API базы данных. Для применения такого драйвера, помимо библиотеки Java, на стороне клиента необходимо установить код, специфический для конкретной платформы.
- **Драйвер типа 3.** Разрабатывается только на основе клиентской библиотеки Java, в которой используется независимый от базы данных протокол передачи запросов базы данных на сервер. Этот протокол приводит запросы базы данных в соответствие с характерным для нее протоколом. Развёртывание прикладных программ значительно упрощается благодаря тому, что код, зависящий от конкретной платформы, находится только на сервере.
- **Драйвер типа 4.** Представляет собой библиотеку, написанную только на Java, для приведения запросов JDBC в соответствие с протоколом конкретной базы данных.



На заметку! Спецификация прикладного интерфейса JDBC доступна для загрузки по адресу http://download.oracle.com/otndocs/jcp/jdbc-4_2-mrel2-spec/.

Большинство поставщиков баз данных предоставляют драйверы типа 3 или 4. Кроме того, целый ряд сторонних производителей специализируется на создании драйверов, которые позволяют добиться более полного соответствия принятым стандартам, поддерживают большее количество платформ, обладают более высокой производительностью или надежностью, чем драйверы, предлагаемые поставщиками баз данных.

Основные цели прикладного интерфейса JDBC можно сформулировать следующим образом.

- Разработчики пишут программы на Java, пользуясь для доступа к базам данных стандартными средствами языка SQL (или его специализированными расширениями), но следуя только соглашениям, принятым в Java.
- Поставщики баз данных и инструментальных средств к ним предоставляют драйверы только низкого уровня. Это дает им возможность оптимизировать драйверы под свою конкретную продукцию.



На заметку! На конференции JavaOne в мае 1996 года представители компании Sun Microsystems указали на ряд следующих причин отказа от модели ODBC.

- Трудна в освоении.
- Имеет всего лишь несколько команд с большим количеством параметров, тогда как стиль программирования на Java основан на применении большого количества простых и интуитивно понятных методов.

- Основана на использовании указателей типа `void*` и других элементов языка С, отсутствующих в Java.
- Менее безопасна и более сложна для развертывания, чем решение, получаемое только на Java.

5.1.2. Типичные примеры применения JDBC

Согласно традиционной модели “клиент–сервер”, графический пользовательский интерфейс (ГПИ) реализуется на стороне клиента, а база данных располагается на стороне сервера (рис. 5.1). В этом случае драйвер JDBC развертывается на стороне клиента.

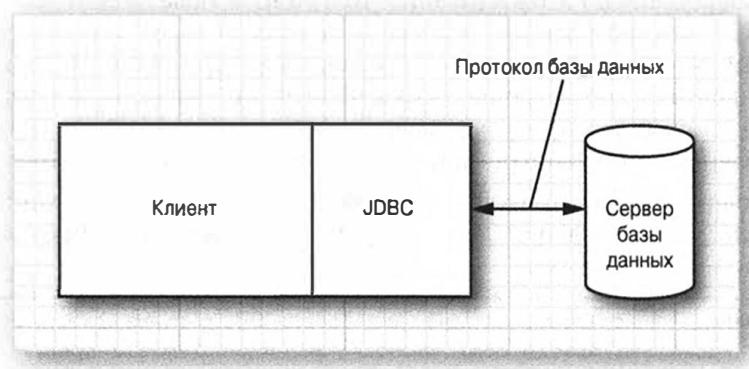


Рис. 5.1. Традиционная структура приложений “клиент–сервер”

Но в настоящее время существует явная тенденция к переходу от архитектуры “клиент–сервер” к трехуровневой модели или даже более совершенной *n*-уровневой модели. В трехуровневой модели клиент не формирует обращения к базе данных. Вместо этого он обращается к средствам промежуточного уровня на сервере, который, в свою очередь, выполняет запросы к базе данных. Трехуровневая модель обладает двумя преимуществами: отделяет *визуальное представление* (на клиентском компьютере) от *бизнес-логики* (на промежуточном уровне) и *исходных данных* (хранящихся в базе данных). Таким образом, становится возможным доступ к тем же самим данным по таким же бизнес-правилам со стороны разнотипных клиентов, в том числе прикладных программ на Java, веб-браузеров и приложений для мобильных устройств.

Взаимодействие клиента и промежуточного уровня может быть реализовано по протоколу HTTP. А прикладной интерфейс JDBC служит для управления взаимодействием промежуточного уровня и серверной базы данных. На рис. 5.2 схематически показана основная архитектура трехуровневой модели.

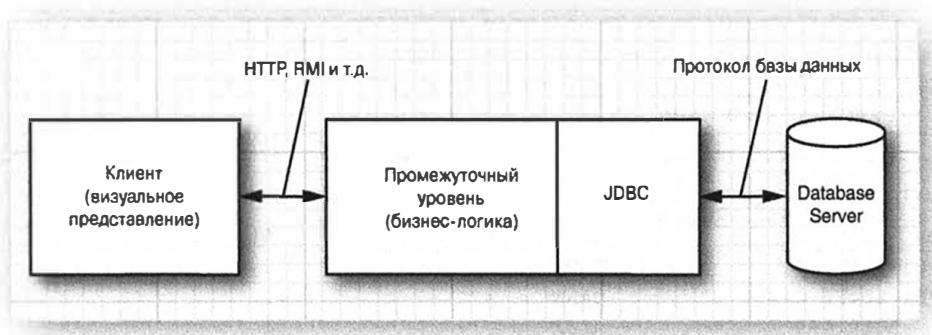


Рис. 5.2. Структура приложений на основе трехуровневой модели

5.2. Язык SQL

Прикладной интерфейс JDBC позволяет взаимодействовать с базами данных с помощью языка SQL, который, в свою очередь, образует интерфейс для большинства современных реляционных баз данных. Настольные базы данных представляют графический интерфейс, который дает пользователям возможность непосредственно манипулировать данными, но доступ к серверным базам данных возможен только с помощью SQL.

Комплект JDBC можно рассматривать лишь как прикладной программный интерфейс API для взаимодействия с командами и операторами языка SQL для доступа к базам данных. В этом разделе приводится краткое описание языка SQL. Если вам не приходилось раньше иметь дело с SQL, то сведений, представленных в этом разделе, может оказаться недостаточно. Для более досконального изучения основ SQL можно порекомендовать книгу *Learning SQL* Алана Болью (Alan Beaulieu; издательство O'Reilly, 2009 г.) или *Learn SQL The Hard Way*, оперативно доступную в электронном виде для заказа по адресу <http://sql.learncodethehardway.org/>.

База данных представляет собой набор именованных таблиц со строками и столбцами. Каждый столбец имеет свое имя, а данные хранятся в строках. В качестве примера базы данных здесь и далее рассматривается ряд таблиц с описаниями библиотеки классических книг по вычислительной технике (табл. 5.1–5.4).

Таблица 5.1. Таблица Authors

Author_ID	Name	Fname
ALEX	Alexander	Christopher
BROO	Brooks	Frederick P.
...

Таблица 5.2. Таблица Books

Title	ISBN	Publisher_ID	Price
A Guide to the SQL Standard	0-201-96426-0	0201	47.95
A Pattern Language: Towns, Buildings, Construction	0-19-501919-9	019	65.00
...

Таблица 5.3. Таблица BooksAuthors

ISBN	Author_ID	Seq_No
0-201-96426-0	DATE	1
0-201-96426-0	DARW	2
0-19-501919-9	ALEX	1
...

Таблица 5.4. Таблица Publishers

Publisher_ID	Name	URL
0201	Addison-Wesley	www.aw-bc.com
0407	John Wiley & Sons	www.wiley.com
...

На рис. 5.3 представлена таблица Books, а на рис. 5.4 — результат соединения таблиц Books и Publishers. Обе таблицы содержат идентификатор издателя. При соединении таблиц по этому коду получается результат запроса в виде таблицы, содержащей данные из обеих исходных таблиц. В каждой строке этой таблицы содержатся сведения о книге, название и адрес веб-сайта издательства. Обратите внимание на то, что данные с названием книги и адресом веб-сайта неоднократно дублируются, поскольку в результирующей таблице оказывается несколько строк, относящихся к одному и тому же издательству.

Title	ISBN	Publisher_ID	Price
The UNIX System Administration Handbook	0-13-020601-6	013	68.00
The C Programming Language	0-13-110362-8	013	42.00
A Pattern Language: Towns, Buildings, Construction	0-19-501919-9	019	65.00
Introduction to Automata Theory, Languages, and Computation	0-201-44124-1	0201	105.00
Design Patterns	0-201-63361-2	0201	54.99
The C++ Programming Language	0-201-70073-5	0201	64.99
The Mythical Man-Month	0-201-83595-9	0201	29.95
Computer Graphics: Principles and Practice	0-201-84840-6	0201	79.99
The Art of Computer Programming vol. 1	0-201-89683-4	0201	59.99
The Art of Computer Programming vol. 2	0-201-89684-2	0201	59.99
The Art of Computer Programming vol. 3	0-201-89685-0	0201	59.99
A Guide to the SQL Standard	0-201-96426-0	0201	47.95
Introduction to Algorithms	0-262-03293-7	0262	80.00
Applied Cryptography	0-471-11709-9	0471	60.00
JavaScript: The Definitive Guide	0-596-00048-0	0596	44.95
The Cathedral and the Bazaar	0-596-00108-8	0596	16.95
The Soul of a New Machine	0-679-60261-5	0679	18.95
The Codebreakers	0-684-83130-9	07434	70.00
Cuckoo's Egg	0-7434-1146-3	07434	13.95
The UNIX Hater's Handbook	1-56884-203-1	0471	16.95

Рис. 5.3. Образец таблицы с данными о книгах

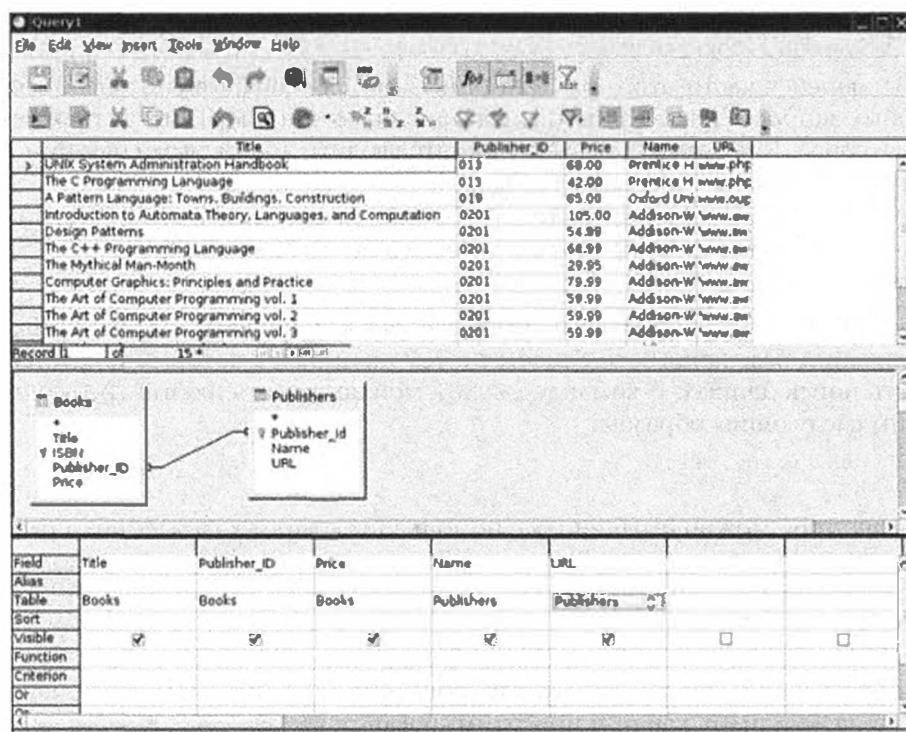


Рис. 5.4. Результат соединения двух таблиц

Преимущество соединения таблиц заключается в том, что в этом случае удастся избежать нежелательного дублирования данных. Например, в простейшей структуре базы данных таблица Books может содержать столбцы с названием и адресом веб-сайта издательства. Но в таком случае данные будут дублироваться уже не только в результате запроса, но и в самой базе данных.

При изменении адреса веб-сайта придется также изменить эти данные во всех записях в базе данных. Очевидно, что при выполнении столь трудоемкой задачи могут легко возникнуть ошибки. В реляционной модели данные распределяются среди нескольких таблиц таким образом, чтобы они не дублировались без особой надобности. Например, адрес веб-сайта каждого издательства хранится в единственном экземпляре в таблице с данными об издательствах. А при необходимости данные из разных таблиц нетрудно соединить в результат запроса.

На рис. 5.3 и 5.4 показано инструментальное средство с ГПИ, предназначенное для просмотра и связывания таблиц. Многие поставщики программного обеспечения предлагают разнообразные диалоговые инструментальные средства для создания запросов путем манипулирования столбцами и ввода данных в готовые формы. Они называются инструментальными средствами составления запросов по образцу (QBE). А при использовании SQL запрос создается в текстовом виде в строгом соответствии с синтаксисом этого языка, как показано ниже.

```
SELECT Books.Title, Books.Publisher_Id, Books.Price,
       Publishers.Name, Publishers.URL
```

```
FROM Books, Publishers  
WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

В оставшейся части этого раздела описываются основные способы создания подобных запросов базы данных. Читатели, знакомые с SQL, могут пропустить этот материал. Ключевые слова SQL принято вводить прописными буквами, хотя это правило не является обязательным.

Команда SELECT может применяться в самых разных целях, например, для выбора всех элементов из таблицы Books по следующему запросу:

```
SELECT * FROM Books
```

Предложение FROM обязательно указывается в каждой команде SELECT. В этом предложении базе данных сообщается о тех таблицах, в которых требуется выполнить поиск данных. В команде SELECT можно указать любые требующиеся столбцы следующим образом:

```
SELECT ISBN, Price, Title  
FROM Books
```

Выбор строк можно ограничить с помощью условия, указываемого в предложении WHERE:

```
SELECT ISBN, Price, Title  
FROM Books  
WHERE Price <= 29.95
```

Следует особо подчеркнуть, что для сравнения в SQL используются операции = и <>, а не == или !=, как при программировании на Java.



На заметку! Некоторые поставщики баз данных используют операцию != для обозначения сравнения, но учтите, что такое обозначение не соответствует стандарту SQL, и поэтому пользоваться им не рекомендуется.

В предложении WHERE может присутствовать оператор LIKE для сопоставления с заданным шаблоном. Но вместо обычных символов подстановки * и ? в данном операторе используется знак %, обозначающий любое количество символов, а знак _ — один символ. Ниже приведен пример запроса на выборку книг, в названиях которых отсутствует такое слово, как UNIX или Linux.

```
SELECT ISBN, Price, Title  
FROM Books  
WHERE Title NOT LIKE '%n_x%'
```

Обратите внимание на то, что в запросах базы данных символьные строки заключаются в одиночные, а не в двойные кавычки. Одиночная кавычка в символьной строке обозначается парой одиночных кавычек, как в приведенном ниже примере запроса на поиск всех книг, в названиях которых содержится одиночная кавычка.

```
SELECT Title  
FROM Books  
WHERE Title LIKE '%''%'
```

Чтобы выбрать данные из нескольких таблиц, их нужно перечислить в следующем порядке:

```
SELECT * FROM Books, Publishers
```

Но без оператора WHERE такой запрос не представляет большого интереса, поскольку по нему получаются все сочетания строк из обеих таблиц. В данном случае таблица Books содержит 20 строк, а таблица Publishers — 8. Поэтому результат выполнения такого запроса будет содержать 20×8 строк с большим количеством дублирующихся данных. Допустим, требуется найти только те книги, которые выпущены издательствами, перечисленными в таблице Publishers. Для поиска такого соответствия книг издательствам можно составить приведенный ниже запрос. Результат выполнения этого запроса содержит 20 строк, т.е. по одной строке на каждую книгу, поскольку на каждую книгу в таблице Publishers приходится лишь одно издательство.

```
SELECT * FROM Books, Publishers  
WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

Если в запросе указано несколько таблиц, то в двух разных местах может упоминаться одно и то же имя столбца, как в показанном выше примере (столбец Publisher_Id из таблицы Books и аналогичный столбец Publisher_Id из таблицы Publishers). Во избежание неоднозначной интерпретации имен столбцов их следует предварять префиксом с именем таблицы, например Books.Publisher_Id.

Языковыми средствами SQL можно пользоваться и для изменения информации в базе данных. Допустим, требуется снизить на 5 долларов текущую цену всех книг, в названиях которых содержится подстрока "C++". С этой целью можно составить следующий запрос:

```
UPDATE Books  
SET Price = Price - 5.00  
WHERE Title LIKE '%C++%'
```

Аналогично для удаления всех книг по C++ понадобится команда DELETE, как показано в приведенном ниже примере запроса. В языке SQL предусмотрены также встроенные функции для вычисления средних значений, поиска максимальных и минимальных значений в столбце и выполнения многих других действий, которые здесь не рассматриваются.

```
DELETE FROM Books  
WHERE Title LIKE '%C++%'
```

Для ввода новых данных в таблицу обычно используется команда INSERT :

```
INSERT INTO Books  
VALUES ('A Guide to the SQL Standard', '0-201-96426-0', '0201', 47.95)
```

Для ввода каждой строки в таблицу приходится выполнять отдельную команду INSERT. Но прежде чем составлять запросы, изменять и вводить данные, необходимо предоставить место для их хранения, т.е. создать таблицу. Для создания новой таблицы служит команда CREATE TABLE, в которой указывается имя и тип данных каждого столбца, как показано в приведенном ниже примере.

```
CREATE TABLE Books  
{  
    Title CHAR(60),  
    ISBN CHAR(13),  
    Publisher_Id CHAR(6),  
    Price DECIMAL(10,2)  
}
```

В табл. 5.5 перечислены наиболее распространенные типы данных в SQL. Дополнительные предложения и операторы, задающие ключи и ограничения, употребляемые в команде CREATE TABLE, здесь не рассматриваются.

Таблица 5.5. Типы данных SQL

Тип данных	Описание
INTEGER или INT	Обычно 32-разрядное целое значение
SMALLINT	Обычно 16-разрядное целое значение
NUMERIC (<i>m</i> , <i>n</i>), DECIMAL (<i>m</i> , <i>n</i>) или DEC (<i>m</i> , <i>n</i>)	Десятичное числовое значение с фиксированной точкой, содержащее <i>m</i> цифр, в том числе <i>n</i> знаков после точки
FLOAT (<i>n</i>)	Числовое значение с плавающей точкой и точностью до <i>n</i> знаков
REAL	Обычно 32-разрядное числовое значение с плавающей точкой
DOUBLE	Обычно 64-разрядное числовое значение с плавающей точкой
CHARACTER (<i>n</i>) или CHAR (<i>n</i>)	Строка фиксированной длины <i>n</i> символов
VARCHAR (<i>n</i>)	Строка переменной длины максимум <i>n</i> символов
BOOLEAN	Логическое значение
DATE	Календарная дата (зависит от реализации)
TIME	Время (зависит от реализации)
TIMESTAMP	Дата и время (зависят от реализации)
BLOB	Большой двоичный объект
CLOB	Большой символьный объект

5.3. Конфигурирование JDBC

Разумеется, для работы с базой данных потребуется система управления базой данных (СУБД), для которой в прикладном интерфейсе JDBC имеется подходящий драйвер. Среди имеющихся СУБД можно выбрать следующие: IBM DB2, Microsoft SQL Server, MySQL, Oracle или PostgreSQL.

Далее необходимо создать экспериментальную базу данных, например, под названием COREJAVA. Создайте новую базу данных сами или попросите сделать это администратора баз данных, а также наделить вас правами для создания, обновления и удаления таблиц.

Если вам не приходилось раньше устанавливать базу данных с архитектурой "клиент–сервер", то процесс ее установки, конечно, покажется вам очень сложным, а обнаружить причину возможной неудачи будет совсем не просто. Поэтому в таких случаях рекомендуется обратиться к услугам опытных специалистов.

Если у вас нет опыта работы с базами данных, установите сначала базу данных Apache Derby, которая входит в состав большинства версий комплекта JDK и доступна для загрузки по адресу <http://db.apache.org/derby>.



На заметку! Версия базы данных Apache Derby, входящая в состав комплекта JDK, официально называется JavaDB. Во избежание недоразумений в этой главе она будет называться Derby.

Прежде чем вы сможете написать свою первую программу для работы с базой данных, вам придется изучить ряд других вопросов, рассматриваемых в последующих разделах.

5.3.1. URL баз данных

Для подключения к базе данных необходимо указать ряд характерных для нее параметров. К их числу могут относиться имена хостов, номера портов, а также имена баз данных. В прикладном интерфейсе JDBC используется синтаксис описания источника данных, подобный обычным URL. Ниже приведены некоторые примеры такого синтаксиса.

```
jdbc:derby://localhost:1527/COREJAVA;create=true  
jdbc:postgresql:COREJAVA
```

Эти URL определяют в JDBC базы данных Derby и PostgreSQL по имени COREJAVA. Ниже приведена общая синтаксическая форма записи URL в JDBC, где подчиненный_протокол обозначает специальный драйвер для соединения с базой данных, а другие_сведения имеют формат, который зависит от применяемого подчиненного протокола. По поводу выбора конкретного формата следует обращаться к документации на применяемую базу данных.

```
jdbc:подчиненный_протокол:другие_сведения
```

5.3.2. Архивные JAR-файлы драйверов

Вам нужно будет также получить архивный JAR-файл, где находится драйвер для выбранной вами базы данных. Так, если вы пользуетесь базой данных Derby, вам понадобится файл derbyclient.jar. Если же это другая база данных, вам придется найти для нее подходящий драйвер. В частности, драйверы для базы данных PostgreSQL доступны для загрузки по адресу <http://jdbc.postgresql.org>.

При запуске программы, обращающейся к базе данных, в командной строке нужно указать архивный JAR-файл драйвера после параметра **-classpath**. (Для компиляции самой программы архивный JAR-файл драйвера не нужен.) Для запуска подобных программ из командной строки можно воспользоваться приведенной ниже командой. В Windows текущий каталог, обозначаемый знаком .., отделяется от местонахождения архивного JAR-файла драйвера точкой с запятой.

```
java -classpath путь_к_файлу_драйвера:.. моя_программа
```

5.3.3. Запуск базы данных

Прежде чем подключиться к серверу базы данных, его нужно запустить. Подробности этого процесса зависят от конкретной базы данных. В частности, для запуска базы данных Derby выполните следующие действия.

1. Откройте командную оболочку и перейдите в каталог, где будут находиться файлы базы данных.
2. Найдите архивный файл derbyrun.jar. В одних версиях JDK он может находиться в каталоге *jdk/db/lib*, а в других – в отдельном установочном каталоге JavaDB. Каталог, содержащий архивный файл *lib/derbyrun.jar*, обозначается здесь и далее как *derby*.
3. Выполните следующую команду:

```
java -jar derby/lib/derbyrun.jar server start
```

4. Еще раз проверьте, работает ли база данных должным образом. Создайте файл `ij.properties`, введя в него следующие строки:

```
ij.driver=org.apache.derby.jdbc.ClientDriver
ij.protocol=jdbc:derby://localhost:1527/
ij.database=COREJAVA;create=true
```

5. Запустите в другой копии командной оболочки диалоговое инструментальное средство для написания сценариев базы данных Derby (оно называется `ij`), выполнив следующую команду:

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties
```

6. Теперь вы можете выдать команды SQL, например, следующие:

```
CREATE TABLE Greetings (Message CHAR(20));
INSERT INTO Greetings VALUES ('Hello, World!');
SELECT * FROM Greetings;
DROP TABLE Greetings;
```

7. Обратите внимание на то, что каждая команда должна завершаться точкой с запятой. Чтобы выйти из режима ввода команд SQL, введите команду `EXIT;`.

8. Завершив работу с базой данных, остановите ее сервер, выполнив следующую команду:

```
java -jar derby/lib/derbyrun.jar server shutdown
```

Если вы пользуетесь другой базой данных, вам нужно найти в документации на нее сведения о запуске и остановке сервера базы данных, а также о том, как подключаться к нему и выполнять команды SQL.

5.3.4. Регистрация класса драйвера

Многие архивные JAR-файлы прикладного интерфейса JDBC (например, драйвер базы данных Derby, входящий в состав версии Java SE 8) автоматически регистрируют класс драйвера. В этом случае вы можете пропустить этап ручной регистрации, рассматриваемый в этом разделе. Архивный JAR-файл может автоматически зарегистрировать класс драйвера, если он содержит файл `META-INF/services/java.sql.Driver`. Чтобы убедиться в этом, достаточно распаковать архивный JAR-файл драйвера.



На заметку! В механизме регистрации используется малоизвестная часть спецификации формата JAR архивных файлов (подробнее об этом см. по адресу <http://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html#Service%20Provider>). Для драйвера, совместимого с версией JDBC4, автоматическая регистрация обязательна.

Если же архивный JAR-файл драйвера не поддерживает автоматическую регистрацию, вам придется выяснить имена классов драйверов JDBC, используемых поставщиком вашей базы данных. Типичными именами классов драйверов являются следующие:

```
org.apache.derby.jdbc.ClientDriver
org.postgresql.Driver
```

Зарегистрировать драйвер с помощью класса `DriverManager` можно двумя способами. Один из них состоит в том, чтобы загрузить класс драйвера в программу на Java, как показано в приведенной ниже строке кода, где выполняется

статический инициализатор, который и осуществляет регистрацию загружаемого драйвера.

```
Class.forName("org.postgresql.Driver");
// принудительно загрузить класс драйвера
```

Другой способ состоит в том, чтобы задать свойство `jdbc.drivers`, которое можно указать в качестве аргумента непосредственно в командной строке:

```
java -Djdbc.drivers=org.postgresql.Driver моя_программа
```

С другой стороны, вы можете установить системное свойство в своей прикладной программе, сделав следующий вызов:

```
System.setProperty("jdbc.drivers", "org.postgresql.Driver");
```

По мере необходимости можно также указать несколько разных драйверов, разделив их двоеточием:

```
org.postgresql.Driver:org.apache.derby.jdbc.ClientDriver
```

5.3.5. Подключение к базе данных

Установить соединение с базой данных в прикладной программе на Java можно следующим образом:

```
String url = "jdbc:postgresql:COREJAVA";
String username = "dbuser";
String password = "secret";
Connection conn =
    DriverManager.getConnection(url, username, password);
```

Диспетчер перебирает все зарегистрированные драйверы, пытаясь найти тот, который соответствует подчиненному протоколу, указанному в URL базы данных. Метод `getConnection()` возвращает объект типа `Connection`, который используется для выполнения команд SQL. Чтобы соединиться с базой данных, необходимо знать имя пользователя базы данных и пароль.



На заметку! По умолчанию база данных Derby допускает соединение под любым именем пользователя, не проверяя пароль. Для каждого пользователя в этой базе данных формируется отдельный ряд таблиц. По умолчанию используется имя пользователя `app`.

Все сказанное выше о работе с базами данных демонстрируется на примере тестовой программы, исходный код которой приведен в листинге 5.1. Эта программа загружает из файла `database.properties` параметры соединения с базой данных и затем устанавливает его. Файл `database.properties`, предоставляемый вместе с примером кода, содержит сведения о соединении с базой данных Derby. Если вы пользуетесь другой базой данных, введите этот в файл соответствующие сведения о соединении с конкретной базой данных. Ниже в качестве примера приведены параметры соединения с базой данных PostgreSQL.

```
jdbc.drivers=org.postgresql.Driver
jdbc.url=jdbc:postgresql:COREJAVA
jdbc.username=dbuser
jdbc.password=secret
```

После соединения с базой данных рассматриваемая здесь тестовая программа выполняет следующие команды SQL:

```
CREATE TABLE Greetings (Message CHAR(20))
INSERT INTO Greetings VALUES ('Hello, World!')
SELECT * FROM Greetings
```

В результате выполнения команды SELECT выводится приведенная ниже символьная строка.

Hello, World!

После этого таблица, созданная в базе данных, удаляется из нее по следующей команде SQL:

```
DROP TABLE Greetings
```

Чтобы выполнить данную тестовую программу, запустите сначала базу данных, как описано выше, а затем саму программу, введя приведенную ниже команду. (Как всегда, пользователям Windows следует ввести точку с запятой (;) вместо двоеточия (:) для разделения составляющих пути к файлу.)

```
java -classpath .:driverJAR test.TestDB
```



Совет! Для устранения неполадок в прикладном интерфейсе JDBC можно активизировать трассировку JDBC. С этой целью вызовите метод `DriverManager.setLogWriter()`, чтобы направить сообщения трассировки в записывающий поток типа `PrintWriter`. Вывод трассировки содержит подробный перечень действий JDBC. В большинстве реализаций драйвера JDBC предоставляются дополнительные механизмы трассировки. Например, для базы данных Derby следует добавить параметр `traceFile` в URL прикладного интерфейса JDBC следующим образом:

```
jdbc:derby://localhost:1527/COREJAVA;create=true;traceFile=trace.out
```

Листинг 5.1. Исходный код из файла test/TestDB.java

```
1 package test;
2
3 import java.nio.file.*;
4 import java.sql.*;
5 import java.io.*;
6 import java.util.*;
7
8 /**
9  * В этой программе проверяется правильность конфигурирования
10 * базы данных и драйвера JDBC
11 * @version 1.02 2012-06-05
12 * @author Cay Horstmann
13 */
14 public class TestDB
15 {
16     public static void main(String args[]) throws IOException
17     {
18         try
19         {
20             runTest();
21         }
22     }
```

```
22     catch (SQLException ex)
23     {
24         for (Throwable t : ex)
25             t.printStackTrace();
26     }
27 }
28
29 /**
30 * Выполняет тест, создавая таблицу, вводя в нее значение,
31 * отображая содержимое таблицы и, наконец, удаляя ее
32 */
33 public static void runTest() throws SQLException, IOException
34 {
35     try (Connection conn = getConnection();
36          Statement stat = conn.createStatement())
37     {
38         stat.executeUpdate("CREATE TABLE Greetings
39                         (Message CHAR(20))");
40         stat.executeUpdate("INSERT INTO Greetings
41                         VALUES ('Hello, World!')");
42
43         try (ResultSet result = stat.executeQuery(
44                           "SELECT * FROM Greetings"))
45         {
46             if (result.next())
47                 System.out.println(result.getString(1));
48         }
49         stat.executeUpdate("DROP TABLE Greetings");
50     }
51 }
52
53 /**
54 * Получает соединение с базой данных из свойств,
55 * определенных в файле database.properties
56 * @return the database connection
57 */
58 public static Connection getConnection()
59     throws SQLException, IOException
60 {
61     Properties props = new Properties();
62     try (InputStream in = Files.newInputStream(
63                           Paths.get("database.properties")))
64     {
65         props.load(in);
66     }
67     String drivers = props.getProperty("jdbc.drivers");
68     if (drivers != null) System.setProperty(
69                     "jdbc.drivers", drivers);
70     String url = props.getProperty("jdbc.url");
71     String username = props.getProperty("jdbc.username");
72     String password = props.getProperty("jdbc.password");
73
74     return DriverManager.getConnection(url, username, password);
75 }
76 }
```

```
java.sql.DriverManager 1.1
```

- **static Connection getConnection(String url, String user, String password)**

Устанавливает соединение с указанной базой данных и возвращает объект типа **Connection**.

5.4. Работа с операторами JDBC

В последующих разделах сначала будет показано, как пользоваться классом **Statement** из прикладного интерфейса JDBC для выполнения операторов SQL, получения результатов и обработки ошибок. А затем будет представлена простая программа для заполнения базы данных.

5.4.1. Выполнение команд SQL

Для выполнения команды SQL сначала создается объект типа **Statement**. Для этой цели используется объект типа **Connection**, который можно получить, вызвав метод **DriverManager.getConnection()** следующим образом:

```
Statement stat = conn.createStatement();
```

Затем формируется символьная строка с требующейся командой SQL, как показано в приведенном ниже примере кода.

```
String command = "UPDATE Books"
    + " SET Price = Price - 5.00"
    + " WHERE Title NOT LIKE '%Introduction%';
```

Далее вызывается метод **executeUpdate()** из класса **Statement**:

```
stat.executeUpdate(command);
```

Метод **executeUpdate()** возвращает количество строк, полученных из таблицы базы данных в результате выполнения команды SQL, или же нуль строк для команд, которые не возвращают количество строк из таблицы. Так, в результате приведенного выше вызова метода **executeUpdate()** возвращается количество книг, цена которых снижена на 5 долларов.

Вызывая метод **executeUpdate()**, можно выполнять команды **INSERT**, **UPDATE** и **DELETE**, а также команды определения данных, в том числе **CREATE TABLE** и **DROP TABLE**. Но для выполнения команды **SELECT** необходимо вызвать другой метод, а именно **executeQuery()**. Имеется также универсальный метод **execute()**, с помощью которого можно выполнять произвольные команды SQL, но он применяется в основном для составления запросов в диалоговом режиме.

Если вы составляете запрос базы данных, вас, конечно, интересует результат его обработки. Метод **executeQuery()** возвращает объект типа **ResultSet**, как показано ниже. Этот объект можно использовать для построчного просмотра результатов выполнения запроса.

```
ResultSet rs = stat.executeQuery("SELECT * FROM Books")
```

Для анализа результирующего набора организуется приведенный ниже цикл.

```

while (rs.next())
{
    проанализировать строку из результирующего набора
}

```



Внимание! Порядок последовательной обработки строк в интерфейсе `ResultSet` организован несколько иначе, чем в интерфейсе `java.util.Iterator`. В интерфейсе `ResultSet` итератор устанавливается на позиции перед первой строкой из результирующего набора. Поэтому для его перемещения к первой строке нужно вызвать метод `next()`. Кроме того, в данном интерфейсе отсутствует метод `hasNext()`, а следовательно, метод `next()` придется вызывать до тех пор, пока не будет возвращено логическое значение `false`.

Строки располагаются в результирующем наборе в совершенно произвольном порядке. Если порядок их следования важен, его необходимо установить с помощью оператора `ORDER BY`. При обработке отдельной строки обычно требуется получить содержимое отдельных полей (или столбцов). Для этой цели имеется целый ряд методов доступа к полям (или столбцам). Ниже приведены некоторые примеры вызова подобных методов доступа.

```

String isbn = rs.getString(1);
double price = rs.getDouble("Price");

```

Для каждого типа данных Java предусмотрен отдельный метод доступа, например `getString()` или `getDouble()`. И каждый из них реализован в двух формах: один — с числовым параметром, другой — со строковым. Если метод доступа вызывается с числовым параметром, данные извлекаются из столбца с указанным номером. Например, в результате вызова метода `rs.getString(1)` возвращается значение из первого столбца текущей строки таблицы.



Внимание! В отличие от массивов, нумерация столбцов таблиц в базе данных начинается с 1.

Если же метод доступа вызывается со строковым параметром, данные извлекаются из столбца с указанным именем. Например, в результате вызова метода `rs.getDouble("Price")` возвращается значение из столбца с именем Price. Первая форма методов доступа с числовым параметром более эффективна, но строковые параметры улучшают восприятие и упрощают сопровождение кода.

А если указанный тип не соответствует фактическому типу, метод доступа автоматически выполняет необходимое преобразование данных. Например, при вызове метода `rs.getString("Price")` числовое значение с плавающей точкой, извлекаемое из столбца Price, преобразуется в символьную строку.

java.sql.Connection 1.1

- **Statement createStatement()**

Создает объект типа `Statement`, который может использоваться для выполнения команд SQL без параметров.

- **void close()**

Немедленно разрывает текущее соединение и освобождает созданные для него ресурсы JDBC.

java.sql.Statement 1.1

- **ResultSet executeQuery(String sqlQuery)**

Выполняет команду SQL из указанной символьной строки и возвращает объект типа **ResultSet** с результатами выполнения этой команды.

- **int executeUpdate(String sqlStatement)**

- **long executeLargeUpdate(String sqlStatement) 8**

Выполняют команды SQL типа **INSERT**, **UPDATE** и **DELETE** из указанной символьной строки, а также команды языка определения данных (DDL) вроде **CREATE TABLE**. Возвращают количество строк, обработанных при выполнении данной команды, или нулевое значение, если для данной команды не установлен подсчет обновлений.

- **boolean execute(String sqlStatement)**

Выполняет команду SQL из указанной символьной строки и возвращает логическое значение **true**, если эта команда предоставляет результирующий набор, а иначе — логическое значение **false**. Может сформировать множество результирующих наборов и подсчетов обновлений. Для доступа к данным, полученным в результате выполнения данной команды SQL, следует вызвать метод **getResultSet()** или **getUpdateCount()**. Вопросы обработки множественных результатов рассматриваются далее в разделе 5.5.4.

- **ResultSet getResultSet()**

Возвращает объект типа **ResultSet** с результатами выполнения предыдущей команды SQL или пустое значение **null**, если выполнение предыдущей команды не дало никаких результатов. Этот метод следует вызывать только один раз для каждой выполняемой команды SQL.

- **int getUpdateCount()**

- **long getLargeUpdateCount() 8**

Возвращают количество строк, обработанных при выполнении предыдущей команды обновления, или значение **-1**, если для данной команды SQL не установлен подсчет обновлений. Эти методы следует вызывать только один раз для каждой выполняемой команды SQL.

- **void close()**

Закрывает данную команду SQL и связанные с ней результирующие наборы.

- **boolean isClosed() 6**

Возвращает логическое значение **true** при закрытии данной команды SQL.

- **void closeOnCompletion() 7**

Закрывает данную команду SQL, как только будут закрыты все связанные с ней результирующие наборы.

java.sql.ResultSet 1.1

- **boolean next()**

Перемещает указатель текущей строки в результирующем наборе на одну позицию вперед. После прохождения последней строки возвращает логическое значение **false**. Следует иметь в виду, что данный метод нужно вызывать для перемещения указателя к первой строке в результирующем наборе.

java.sql.ResultSet 1.1 (окончание)

- **Xxx getXxx(int columnNumber)**
 - **Xxx getXxx(String columnName)**
 - (**Xxx** обозначает тип данных, например **int, double, String, Date** и т.д.)
 - **<T> T getObject(int columnNumber, Class<T> type) 7**
 - **<T> T getObject(String columnName, Class<T> type) 7**
- Возвращает значение столбца, задаваемого номером или меткой, с преобразованием в указанный тип данных. Следует иметь в виду, что допускаются не все варианты преобразования типов. Метка столбца — это метка, указываемая в предложении **AS** команды SQL, или имя столбца, если предложение **AS** не используется.
- **int findColumn(String columnName)**
Возвращает номер столбца с указанным именем.
 - **void close()**
Немедленно закрывает текущий результирующий набор.
 - **boolean isClosed() 6**
Возвращает логическое значение **true** при закрытии данной команды.

5.4.2. Управление соединениями, командами и результирующими наборами

Каждый объект типа **Connection** может создать один или несколько объектов типа **Statement**. Один и тот же объект типа **Statement** можно использовать для выполнения нескольких не связанных между собой команд и запросов. Но для такого объекта допускается наличие *не более одного открытого результирующего набора*. Если же требуется выполнить несколько команд и одновременно проанализировать их результаты, то для этого понадобится несколько объектов типа **Statement**.

Не следует, однако, забывать, что, по крайней мере, одна широко употребляемая база данных (Microsoft SQL Server) взаимодействует с драйвером JDBC, допускающим одновременную активизацию только одного объекта типа **Statement**. Количество открытых объектов типа **Statement**, одновременно поддерживаемых драйвером JDBC, можно выяснить, вызвав метод **getMaxStatements()** из класса **DatabaseMetaData**.

На первый взгляд, подобное ограничение кажется излишним, но на практике дело редко доходит до одновременной обработки нескольких результирующих наборов. Если же результирующие наборы связаны друг с другом, то можно всегда сформировать составной запрос и проанализировать единственный результирующий набор. Намного выгоднее предоставить базе данных возможность самостоятельно объединить запросы, чем циклически обрабатывать несколько результирующих наборов в программе на Java.

Покончив дело с объектом типа **ResultSet**, **Statement** или **Connection**, следует как можно скорее вызвать метод **close()**. Ведь эти объекты используют крупные структуры данных и источаемые ресурсы на сервере базы данных. Метод **close()** из класса **Statement** автоматически закрывает результирующий набор, связанный с объектом данного класса, если, конечно, этот набор открыт

при выполнении соответствующей команды. Аналогично метод `close()` из класса `Connection` закрывает все объекты данного класса, открытые для соединения с базой данных. С другой стороны, в версии Java SE 7 появилась возможность вызвать метод `closeOnCompletion()` из класса `Statement`, чтобы автоматически закрыть объект данного класса, как только будут закрыты все связанные с ним результирующие наборы.

Если соединение установлено кратковременно, то можно и не беспокоиться о закрытии объектов типа `Statement` и связанных с ними результирующих наборов. А для абсолютной гарантии, что объект соединения с базой данных не останется открытым, можно воспользоваться оператором `try` с ресурсами следующим образом:

```
try (Connection conn = . . .)
{
    Statement stat = conn.createStatement();
    ResultSet result = stat.executeQuery(queryString);
    обработать результат запроса
}
```



Совет! Оператор `try` с ресурсами рекомендуется применять только для разрыва соединения с базой данных, а для обработки исключений — отдельный блок операторов `try/catch`. Благодаря такому разделению блоков операторов исходный код легче читать и сопровождать.

5.4.3. Анализ исключений SQL

У каждого исключения SQL имеется цепочка объектов типа `SQLException`, которые извлекаются методом `getNextException()`. Эта цепочка исключений является дополнением “причинной” цепочки объектов типа `Throwable`, имеющихся в каждом исключении. (Подробнее об исключениях в Java см. в главе 11 первого тома данной книги.) Чтобы полностью перечислить все исключения, пришлось бы организовать два вложенных цикла. К счастью, в версии Java SE 6 был усовершенствован класс `SQLException` для реализации интерфейса `Iterable<Throwable>`. В частности, метод `iterator()` производит объект типа `Iterator<Throwable>`, который осуществляет перебор в обеих цепочках, сначала проходя по “причинной” цепочке первого объекта типа `SQLException`, а затем переходя к следующему объекту типа `SQLException` и т.д. Для этой цели можно организовать усовершенствованный цикл `for` следующим образом:

```
for (Throwable t : sqlException)
{
    сделать что-нибудь с объектом t
}
```

Чтобы продолжить анализ объекта типа `SQLException`, для него можно вызвать методы `getSQLState()` и `getErrorCode()`. Первый метод выдает символьную строку по стандарту X/Open или SQL:2003. (Чтобы выяснить, какому именно стандарту соответствует применяемый драйвер, достаточно вызвать метод `getSQLStateType()` из интерфейса `DatabaseMetadata`.) Что же касается кода ошибки, то у различных поставщиков баз данных он разный.

Исключения SQL организованы в виде древовидной структуры наследования, приведенной на рис. 5.5. Благодаря этому имеется возможность перехватывать отдельные типы ошибок независимо от предпочтений поставщиков баз данных.

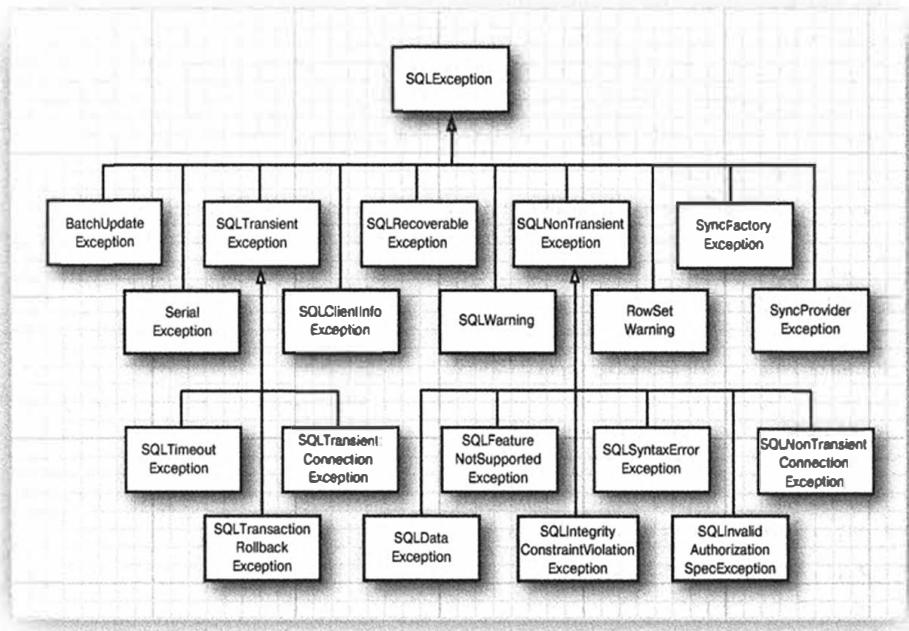


Рис. 5.5. Типы исключений SQL

Кроме того, драйвер базы данных может сообщать о некритичных ситуациях в виде предупреждений. Подобные предупреждения можно получать от соединений, команд и результирующих наборов. Класс `SQLWarning` является подклассом, производным от класса `SQLException`, несмотря на то, что объект типа `SQLWarning` не генерируется в виде исключения. Вызвав методы `getSQLState()` и `getErrorCode()`, можно получить дополнительные сведения о предупреждениях. Подобно исключениям SQL, предупреждения организуются в цепочку. И чтобы получить все предупреждения, придется организовать следующий цикл:

```

SQLWarning w = stat.getWarning();
while (w != null)
{
    сделать что-нибудь с объектом w
    w = w.nextWarning();
}
  
```

Подкласс `DataTruncation`, производный от класса `SQLWarning`, используется в тех случаях, когда данныечитываются из базы данных и в этот момент происходит неожиданное их усечение. Если усечение данных произошло при выполнении команды обновления, то объект типа `DataTruncation` генерируется в виде исключения.

java.sql.SQLException 1.1

- **SQLException getNextException()**

Получает исключение SQL, следующее за данным исключением по цепочке, или пустое значение `null`, если достигнут конец цепочки.

- **Iterator<Throwable> iterator() 6**

Получает итератор, перебирающий по цепочке исключения SQL и их причины.

- **String getSQLState()**

Получает стандартный код ошибки, обозначающий состояние SQL.

- **int getErrorCode()**

Получает код ошибки, характерный для поставщика используемой базы данных.

java.sql.SQLWarning 1.1

- **SQLWarning getNextWarning()**

Получает предупреждение, следующее за данным исключением по цепочке, или пустое значение `null`, если достигнут конец цепочки.

java.sql.Connection 1.1**java.sql.Statement 1.1****java.sql.ResultSet 1.1**

- **SQLWarning getWarnings()**

Возвращает первое ожидающее предупреждение или пустое значение `null`, если ожидающие предупреждения отсутствуют.

java.sql.DataTruncation 1.1

- **boolean getParameter()**

Возвращает логическое значение `true`, если усечение данных применяется к параметру, или логическое значение `false`, если оно применяется к столбцу.

- **int getIndex()**

Возвращает индекс усеченного параметра или столбца.

- **int getDataSize()**

Возвращает количество байтов, которые необходимо передать, или значение `-1`, если извлекаемое значение неизвестно.

- **int getTransferSize()**

Возвращает количество байтов, которые были фактически переданы, или значение `-1`, если извлекаемое значение неизвестно.

5.4.4. Заполнение базы данных

Попробуем теперь написать первую программу, используя прикладной интерфейс JDBC. Конечно, было бы неплохо, если бы в этой программе можно было выполнить некоторые из рассмотренных выше запросов. Но, к сожалению, это невозможно, потому что база данных пуста. Сначала ее нужно заполнить данными, хотя сделать это нетрудно с помощью команд SQL для создания таблиц и ввода в них данных. Большинство СУБД способны обрабатывать команды SQL из текстового файла, но в этом случае проявляются досадные отличия в завершающих символах операторов и другие синтаксические особенности реализации SQL на разных платформах.

В силу этих причин воспользуемся прикладным интерфейсом JDBC, чтобы создать простую программу для построчного чтения команд SQL из файла и последующего их выполнения. В частности, рассматриваемая здесь программа должна читать данные из текстового файла в следующем формате:

```
CREATE TABLE Publishers (Publisher_Id CHAR(6), Name CHAR(30),
                         URL CHAR(80));
INSERT INTO Publishers VALUES ('0201', 'Addison-Wesley',
                               'www.aw-bc.com');
INSERT INTO Publishers VALUES ('0471', 'John Wiley & Sons',
                               'www.wiley.com');
...
```

В листинге 5.2 приведен исходный код программы ExecSQL,читывающей команды SQL из текстового файла и затем выполняющей их. Для применения этой программы совсем не обязательно разбираться в ее исходном коде. Самое главное, что она позволяет заполнить базу данных и выполнить примеры программ в оставшейся части этой главы.

Прежде всего убедитесь в том, что сервер базы данных работает нормально, и запустите программу ExecSQL на выполнение, введя следующие команды:

```
java -classpath путь_к_драйверу:. exec.ExecSQL Books.sql
java -classpath путь_к_драйверу:. exec.ExecSQL Authors.sql
java -classpath путь_к_драйверу:. exec.ExecSQL Publishers.sql
java -classpath путь_к_драйверу:. exec.ExecSQL BooksAuthors.sql
```

Перед запуском данной программы обязательно проверьте содержимое файла свойств database.properties на соответствие вашей исполняющей среде (см. раздел 5.3.5 ранее в этой главе).



На заметку! В состав вашей базы данных может входить утилита для непосредственного чтения файлов SQL. Например, в базе данных Derby можно выполнить приведенную ниже команду для чтения файла свойств ij.properties, описанного ранее в разделе 5.3.3.

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties Books.sql
```

В формате данных для программы ExecSQL допускается вставка точки с запятой в конце каждой строки, поскольку такой формат предполагается в большинстве утилит баз данных.

Ниже вкратце описываются основные этапы выполнения программы ExecSQL.

1. Устанавливается соединение с базой данных. Метод getConnection() считывает содержимое файла свойств database.properties и вводит свойство

`jdbc.drivers` в список системных свойств. Диспетчер драйверов использует свойство `jdbc.drivers` для загрузки соответствующего драйвера базы данных. Для подключения к базе данных в методе `getConnection()` применяются свойства `jdbc.url`, `jdbc.username` и `jdbc.password`.

2. Открывается текстовый файл с командами SQL. Если такой файл отсутствует, пользователю предлагается ввести команды вручную с консоли.
3. Все заданные команды SQL выполняются с помощью универсального метода `execute()`. При получении результирующего набора этот метод возвращает логическое значение `true`. В конце всех четырех текстовых файлов с командами SQL содержится команда `SELECT *.` Это дает возможность убедиться, что данные успешно введены в базу данных.
4. При наличии результирующего набора полученные результаты выводятся на экран. А поскольку это обобщенный результирующий набор, то для определения количества столбцов в нем потребуются метаданные. Более подробно метаданные рассматриваются далее, в разделе 5.8.
5. Если при выполнении команд SQL возникает какое-нибудь исключение, то выводятся сведения о нем, а также обо всех остальных исключениях, которые могут следовать по цепочке.
6. По завершении всех заданных команд SQL соединение с базой данных разрывается.

Листинг 5.2. Исходный код из файла exec/ExecSQL.java

```
1 package exec;
2
3 import java.io.*;
4 import java.nio.file.*;
5 import java.util.*;
6 import java.sql.*;
7
8 /**
9  * Эта программа выполняет все команды SQL из текстового файла.
10 * Она вызывается следующим образом:
11 * java -classpath путь_к_драйверу:. ExecSQL командный_файл
12 *
13 * @version 1.32 2016-04-27
14 * @author Cay Horstmann
15 */
16 class ExecSQL
17 {
18     public static void main(String args[]) throws IOException
19     {
20         try (Scanner in = args.length == 0 ? new Scanner(System.in)
21              : new Scanner(Paths.get(args[0]), "UTF-8"))
22         {
23             try (Connection conn = getConnection();
24                  Statement stat = conn.createStatement())
25             {
26                 while (true)
27                 {
```

```
28         if (args.length == 0)
29             System.out.println("Enter command or EXIT to exit:");
30
31         if (!in.hasNextLine()) return;
32
33         String line = in.nextLine().trim();
34         if (line.equalsIgnoreCase("EXIT")) return;
35         if (line.endsWith(";")) // удалить точку
36             // с запятой в конце строки
37         {
38             line = line.substring(0, line.length() - 1);
39         }
40         try
41         {
42             boolean isResult = stat.execute(line);
43             if (isResult)
44             {
45                 try (ResultSet rs = stat.getResultSet())
46                 {
47                     showResultSet(rs);
48                 }
49             }
50             else
51             {
52                 int updateCount = stat.getUpdateCount();
53                 System.out.println(updateCount + " rows updated");
54             }
55         }
56         catch (SQLException ex)
57         {
58             for (Throwable e : ex)
59                 e.printStackTrace();
60         }
61     }
62 }
63 }
64 catch (SQLException e)
65 {
66     for (Throwable t : e)
67         t.printStackTrace();
68 }
69 }
70 /**
71 * Получает соединение с базой данных из свойств,
72 * определенных в файле database.properties
73 * @return Возвращает соединение с базой данных
74 */
75 public static Connection getConnection()
76     throws SQLException, IOException
77 {
78     Properties props = new Properties();
79     try (InputStream in = Files.newInputStream(Paths.get(
80                                     "database.properties")))
81     {
82         props.load(in);
83     }
84 }
```

```
86     String drivers = props.getProperty("jdbc.drivers");
87     if (drivers != null)
88         System.setProperty("jdbc.drivers", drivers);
89
90     String url = props.getProperty("jdbc.url");
91     String username = props.getProperty("jdbc.username");
92     String password = props.getProperty("jdbc.password");
93
94     return DriverManager.getConnection(url, username, password);
95 }
96
97 /**
98 * Выводит результирующий набор
99 * @param result Возвращает выводимый результирующий набор
100 */
101 public static void showResultSet(ResultSet result) throws SQLException
102 {
103     ResultSetMetaData metaData = result.getMetaData();
104     int columnCount = metaData.getColumnCount();
105
106     for (int i = 1; i <= columnCount; i++)
107     {
108         if (i > 1) System.out.print(", ");
109         System.out.print(metaData.getColumnLabel(i));
110     }
111     System.out.println();
112
113     while (result.next())
114     {
115         for (int i = 1; i <= columnCount; i++)
116         {
117             if (i > 1) System.out.print(", ");
118             System.out.print(result.getString(i));
119         }
120         System.out.println();
121     }
122 }
123 }
```

5.5. Выполнение запросов

В этом разделе рассматривается пример программы, способной выполнять запросы базы данных COREJAVA. Для нормальной работы этой программы в базе данных нужно создать таблицы, описанные в предыдущем разделе. При составлении запроса базы данных можно выбрать автора книги и издательство или же оставить критерий отбора книги независимо от автора или издательства.

Рассматриваемая здесь программа позволяет также вносить изменения в содержимое базы данных. Для этого достаточно выбрать издательство и ввести сумму. Все цены на книги данного издательства автоматически откорректируются по введенной сумме, а программа отобразит количество измененных строк в таблице. После подобной коррекции цен на книги можно выполнить запрос, чтобы проверить новые цены.

5.5.1. Подготовленные операторы и запросы

В рассматриваемой здесь программе используется новое средство: *подготовленные операторы*. Рассмотрим следующий запрос SQL на выборку всех книг отдельного издательства независимо от их авторов:

```
SELECT Books.Price, Books.Title  
FROM Books, Publishers  
WHERE Books.Publisher_Id = Publishers.Publisher_Id  
AND Publishers.Name = название издательства, выбираемое из списка
```

Вместо создания отдельной команды SQL для каждого пользовательского запроса можно заранее *подготовить* запрос с главной переменной и многократно использовать его, меняя только значение этой переменной. Такая возможность существенно повышает эффективность работы программы. Перед обработкой каждого запроса СУБД вырабатывает план его эффективного исполнения. Подготавливая запрос для последующего многократного применения, можно исключить повторное планирование его выполнения.

Каждая главная переменная в запросе обозначается знаком вопроса (?). Если в запросе используются несколько главных переменных, необходимо внимательно следить за их расстановкой с помощью знаков вопроса, чтобы правильно устанавливать их конкретные значения. Ниже показано, как выглядит предварительно подготовленный запрос рассматриваемой здесь базы данных изданий в исходном коде.

```
String publisherQuery =  
    "SELECT Books.Price, Books.Title" +  
    " FROM Books, Publishers" +  
    " WHERE Books.Publisher_Id = Publishers.Publisher_Id  
        AND Publishers.Name = ?";  
PreparedStatement stat = conn.prepareStatement(publisherQuery);
```

Перед выполнением подготовленного оператора необходимо связать главные переменные с их конкретными значениями, вызвав метод set(). Подобно разным формам метода get() из интерфейса ResultSet, для разных типов данных предусмотрены отдельные формы метода set(). В качестве примера ниже показано, каким образом задается строковое значение с названием издательства.

```
stat.setString(1, publisher);
```

Первый аргумент этого метода обозначает номер позиции главной переменной, обозначаемой знаком вопроса в подготовленном операторе, а второй аргумент — ее конкретное значение.

При повторном использовании подготовленного запроса с несколькими главными переменными все их привязки к конкретным значениям остаются в силе, если только они не изменены с помощью метода set() или clearParameters(). Это означает, что метод setXxx(), где Xxx — тип данных, нужно вызывать только для тех главных переменных, которые изменяются в последующих запросах.

После привязки всех переменных к их конкретным значениям можно приступить к выполнению подготовленного оператора следующим образом:

```
ResultSet rs = stat.executeQuery();
```



Совет Составление запроса вручную путем склеивания символьных строк — довольно трудоемкое, чреватое ошибками и небезопасное занятие. Ведь в этом случае нужно позаботиться об обозначении специальных символов (например, кавычек). А если при составлении запроса предполагается ввод пользователем данных, необходимо принять меры защиты от умышленного внесения запросов SQL при совершении атак на сервер базы данных. В этом отношении подготовленные операторы оказываются намного более удобными, и поэтому их рекомендуется применять всякий раз, когда в запрос включаются переменные.

Обновление цены осуществляется по команде UPDATE. Обратите внимание на то, что для этого вызывается метод executeUpdate(), а не executeQuery(). Дело в том, что команда UPDATE не возвращает результирующий набор, который в данном случае не нужен. Метод executeUpdate() возвращает лишь подсчет количества измененных строк в таблице, как показано ниже.

```
int r = stat.executeUpdate();
System.out.println(r + " rows updated");
```



На заметку! Объект типа PreparedStatement становится недействительным после того, как связанный с ним объект типа Connection закрывается. Но многие драйверы баз данных автоматически кешируют подготовленные операторы. Если один и тот же запрос подготавливается дважды, то в СУБД просто еще раз используется план его выполнения. Поэтому, вызывая метод prepareStatement(), можно не особенно беспокоиться об издержках на выполнение подготовленных операторов.

Ниже вкратце описывается порядок действий, выполняемых в рассматриваемом здесь примере программы.

- Списочные массивы заполняются именами авторов и названиями издательств по двум запросам, из которых возвращаются все имена авторов и названия издательств, сохраняемые в базе данных.
- Запросы по имени автора имеют более сложную структуру. Ведь у одной книги может быть несколько авторов, и поэтому в таблице BooksAuthors сохраняется соответствие авторов и книг. Допустим, у книги с ISBN 0-201-96426-0 два автора с кодами DATE и DARW. Для отражения этого факта таблица BooksAuthors должна содержать следующие строки:

```
0-201-96426-0, DATE, 1
0-201-96426-0, DARW, 2
```

- В третьем столбце указаны порядковые номера авторов. (Для этой цели нельзя использовать сведения о расположении строк в таблице, поскольку в реляционной базе данных порядок следования записей не фиксирован.) Таким образом, в составляемом запросе следует сначала соединить таблицы Books, BooksAuthors и Authors, а затем сравнить в них имя автора с тем, что указано пользователем:

```
SELECT Books.Price, Books.Title FROM Books, BooksAuthors,
    Authors, Publishers
WHERE Authors.Author_Id =
    BooksAuthors.Author_Id AND BooksAuthors.ISBN = Books.ISBN
AND Books.Publisher_Id =
    Publishers.Publisher_Id AND Authors.Name = ? AND Publishers.
Name = ?
```



На заметку! Некоторые программирующие на Java стараются избегать составления столь сложных запросов SQL. Как ни странно, они выбирают обходной, но неэффективный путь, предполагающий написание немалого объема кода на Java для последовательной обработки нескольких результирующих наборов. Следует, однако, иметь в виду, что СУБД выполняет запросы намного эффективнее, чем программа на Java, поскольку СУБД именно для этого и предназначена. В этой связи рекомендуется взять на вооружение следующее эмпирическое правило: то, что можно сделать средствами SQL, нецелесообразно делать средствами Java.

- Метод changePrices () выполняет команду UPDATE. В предложении WHERE этой команды требуется указать код издательства, а известно лишь его *название*. Это затруднение разрешается с помощью вложенного запроса, как выделено ниже полужирным.

```
UPDATE Books
SET Price = Price + ?
WHERE Books.Publisher_Id =
    (SELECT Publisher_Id FROM Publishers WHERE Name = ?)
```

Весь исходный код данной программы приведен в листинге 5.3.

Листинг 5.3. Исходный код из файла query/QueryTest.java

```
1  package query;
2
3  import java.io.*;
4  import java.nio.file.*;
5  import java.sql.*;
6  import java.util.*;
7
8  /**
9   * В этой программе демонстрируется ряд сложных
10  * запросов базы данных
11  * @version 1.30 2012-06-05
12  * @author Cay Horstmann
13  */
14 public class QueryTest
15 {
16     private static final String allQuery =
17         "SELECT Books.Price, Books.Title FROM Books";
18
19     private static final String authorPublisherQuery =
20         "SELECT Books.Price, Books.Title"
21         + " FROM Books, BooksAuthors, Authors, Publishers"
22         + " WHERE Authors.Author_Id = BooksAuthors.Author_Id"
23         + " AND BooksAuthors.ISBN = Books.ISBN"
24         + " AND Books.Publisher_Id = Publishers.Publisher_Id"
25         + " AND Authors.Name = ?"
26         + " AND Publishers.Name = ?";
27
28     private static final String authorQuery
29         = "SELECT Books.Price, Books.Title FROM Books,
30             BooksAuthors, Authors"
31         + " WHERE Authors.Author_Id = BooksAuthors.Author_Id"
32         + " AND BooksAuthors.ISBN = Books.ISBN"
33         + " AND Authors.Name = ?";
34 }
```

```
35     private static final String publisherQuery
36         = "SELECT Books.Price, Books.Title FROM Books, Publishers"
37         + " WHERE Books.Publisher_Id = Publishers.Publisher_Id
38             AND Publishers.Name = ?";
39
40
41     private static final String priceUpdate = "UPDATE Books "
42         + "SET Price = Price + ?"
43         + " WHERE Books.Publisher_Id =
44             (SELECT Publisher_Id FROM Publishers WHERE Name = ?)";
45
46     private static Scanner in;
47     private static ArrayList<String> authors = new ArrayList<>();
48     private static ArrayList<String> publishers = new ArrayList<>();
49
50     public static void main(String[] args) throws IOException
51     {
52         try (Connection conn = getConnection())
53     {
54             in = new Scanner(System.in);
55             authors.add("Any");
56             publishers.add("Any");
57             try (Statement stat = conn.createStatement())
58             {
59                 // заполнить списочный массив именами авторов книг
60                 String query = "SELECT Name FROM Authors";
61                 try (ResultSet rs = stat.executeQuery(query))
62                 {
63                     while (rs.next())
64                         authors.add(rs.getString(1));
65                 }
66
67                 // заполнить списочный массив названиями издательств
68                 query = "SELECT Name FROM Publishers";
69                 try (ResultSet rs = stat.executeQuery(query))
70                 {
71                     while (rs.next())
72                         publishers.add(rs.getString(1));
73                 }
74             }
75             boolean done = false;
76             while (!done)
77             {
78                 System.out.print("Q)uerie C)hange prices E)xit: ");
79                 String input = in.nextLine().toUpperCase();
80                 if (input.equals("Q"))
81                     executeQuery(conn);
82                 else if (input.equals("C"))
83                     changePrices(conn);
84                 else
85                     done = true;
86             }
87         }
88         catch (SQLException e)
89         {
90             for (Throwable t : e)
91                 System.out.println(t.getMessage());
92         }
93     }
```

```
93     }
94
95     /**
96      * Выполняет выбранный запрос
97      * @param conn Соединение с базой данных
98      */
99     private static void executeQuery(Connection conn)
100        throws SQLException
101    {
102        String author = select("Authors:", authors);
103        String publisher = select("Publishers:", publishers);
104        PreparedStatement stat;
105        if (!author.equals("Any") && !publisher.equals("Any"))
106        {
107            stat = conn.prepareStatement(authorPublisherQuery);
108            stat.setString(1, author);
109            stat.setString(2, publisher);
110        }
111        else if (!author.equals("Any") && publisher.equals("Any"))
112        {
113            stat = conn.prepareStatement(authorQuery);
114            stat.setString(1, author);
115        }
116        else if (author.equals("Any") && !publisher.equals("Any"))
117        {
118            stat = conn.prepareStatement(publisherQuery);
119            stat.setString(1, publisher);
120        }
121        else
122            stat = conn.prepareStatement(allQuery);
123
124        try (ResultSet rs = stat.executeQuery())
125        {
126            while (rs.next())
127                System.out.println(rs.getString(1)
128                                + ", " + rs.getString(2));
129        }
130    }
131
132    /**
133     * Выполняет команду обновления с целью изменить цены на книги
134     * @param conn Соединение с базой данных
135     */
136    public static void changePrices(Connection conn)
137        throws SQLException
138    {
139        String publisher = select("Publishers:",
140            publishers.subList(1, publishers.size()));
141        System.out.print("Change prices by: ");
142        double priceChange = in.nextDouble();
143        PreparedStatement stat =
144            conn.prepareStatement(priceUpdate);
145        stat.setDouble(1, priceChange);
146        stat.setString(2, publisher);
147        int r = stat.executeUpdate();
148        System.out.println(r + " records updated.");
149    }
150}
```

```

151 /**
152 * Предлагает пользователю выбрать символьную строку
153 * @param prompt Отображаемое приглашение
154 * @param options Варианты выбора, предлагаемые пользователю
155 * @return Возвращает выбранный пользователем вариант
156 */
157 public static String select(String prompt, List<String> options)
158 {
159     while (true)
160     {
161         System.out.println(prompt);
162         for (int i = 0; i < options.size(); i++)
163             System.out.printf("%2d %s%n", i + 1, options.get(i));
164         int sel = in.nextInt();
165         if (sel > 0 && sel <= options.size())
166             return options.get(sel - 1);
167     }
168 }
169 /**
170 * Получает соединение с базой данных из свойств,
171 * определенных в файле database.properties
172 * @return Возвращает соединение с базой данных
173 */
174 public static Connection getConnection()
175     throws SQLException, IOException
176 {
177     Properties props = new Properties();
178     try (InputStream in =
179          Files.newInputStream(Paths.get("database.properties")))
180     {
181         props.load(in);
182     }
183
184     String drivers = props.getProperty("jdbc.drivers");
185     if (drivers != null)
186         System.setProperty("jdbc.drivers", drivers);
187     String url = props.getProperty("jdbc.url");
188     String username = props.getProperty("jdbc.username");
189     String password = props.getProperty("jdbc.password");
190
191     return DriverManager.getConnection(url, username,
192                                         password);
193 }
194 }
195 }
```

java.sql.Connection 1.1

- **PreparedStatement prepareStatement(String sql)**

Возвращает объект типа **PreparedStatement**, содержащий подготовленный оператор. Заданная строка **sql** содержит оператор SQL с одной или несколькими заполнителями главных переменных, обозначенными вопросительными знаками.

java.sql.PreparedStatement 1.1

- **void setXxx(int n, Xxx x)**
 (Xxx обозначает тип данных, например **int**, **double**, **String**, **Date** и т.д.)
 Задает значение x для n-го параметра.
- **void clearParameters()**
 Очищает все текущие параметры в подготовленном операторе.
- **ResultSet executeQuery()**
 Выполняет подготовленный запрос SQL и возвращает объект типа **ResultSet**.
- **int executeUpdate()**
 Выполняет команды **INSERT**, **UPDATE** или **DELETE**, представленные в объекте типа **PreparedStatement** в виде подготовленных операторов SQL. Возвращает количество обработанных строк или нулевое значение для таких команд языка DDL, как **CREATE TABLE**.

5.5.2. Чтение и запись больших объектов

Помимо чисел, символьных строк и дат, во многих базах данных можно сохранять *большие объекты* (LOB), к числу которых относятся изображения и другие данные. В языке SQL понятие больших объектов разделяется на категории больших двоичных объектов (BLOB) и больших символьных объектов (CLOB).

Чтобы прочитать большой объект, необходимо сначала выполнить команду **SELECT**, а затем вызвать метод **getBlob()** или **getBlob()** из интерфейса **ResultSet**. В результате будет получен объект типа **Blob** или **Clob**. А для того чтобы получить двоичные данные из объекта типа **Blob**, следует вызвать метод **getBytes()** или **getInputStream()**. Так, если имеется таблица с изображениями на книжных обложках, то такое изображение можно получить следующим образом:

```
PreparedStatement stat =
    conn.prepareStatement("SELECT Cover FROM BookCovers WHERE ISBN=?");
...
stat.set(1, isbn);
ResultSet result = stat.executeQuery();
if (result.next())
{
    Blob coverBlob = result.getBlob(1);
    Image coverImage = ImageIO.read(coverBlob.getBinaryStream());
}
```

Аналогично, если извлечь объект типа **Clob**, то из него можно получить символьные данные, вызвав метод **String()** или **getCharacterStream()**.

Чтобы разместить большой объект в базе данных, следует вызвать метод **createLOB()** или **createClob()** для объекта типа **Connection**, получить поток вывода или поток записи для большого объекта, записать данные и сохранить этот объект в базе данных. В качестве примера ниже показано, как сохранить изображение в базе данных.

```
Blob coverBlob = connection.createBlob();
int offset = 0;
OutputStream out = coverBlob.setBinaryStream(offset);
```

```
ImageIO.write(coverImage, "PNG", out);
PreparedStatement stat =
    conn.prepareStatement("INSERT INTO Cover VALUES (?, ?)");
stat.set(1, isbn);
stat.set(2, coverBlob);
stat.executeUpdate();
```

java.sql.ResultSet 1.1

- **Blob getBlob(int columnIndex) 1.2**
- **Blob getBlob(String columnLabel) 1.2**
- **Clob getClob(int columnIndex) 1.2**
- **Clob getClob(String columnLabel) 1.2**

Получают большой двоичный объект (BLOB) или большой символьный объект (CLOB) из заданного столбца таблицы.

java.sql.Blob 1.2

- **long length()**
Получает длину данного большого двоичного объекта.
- **byte[] getBytes(long startPosition, long length)**
Получает данные в указанных пределах из текущего большого двоичного объекта.
- **InputStream getBinaryStream()**
- **InputStream getBinaryStream(long startPosition, long length)**
Возвращают поток ввода для чтения данных из текущего большого двоичного объекта полностью или в указанных пределах.
- **OutputStream setBinaryStream(long startPosition) 1.4**
Возвращает поток вывода для записи данных в текущий большой двоичный объект, начиная с указанной позиции.

java.sql.Clob 1.4

- **long length()**
Получает количество символов в данном большом символьном объекте.
- **String getSubString(long startPosition, long length)**
Получает символы из текущего большого двоичного объекта в указанных пределах.
- **Reader getCharacterStream()**
- **Reader getCharacterStream(long startPosition, long length)**
Возвращают поток чтения (а не поток ввода) символов из данного большого символьного объекта в указанных пределах.
- **Writer setCharacterStream(long startPosition) 1.4**
Возвращает поток записи (а не поток вывода) символов в данный большой символьный объект, начиная с указанной позиции.

```
java.sql.Connection 1.1
```

- Blob createBlob() 6
- Clob createClob() 6

Создает пустым большой двоичный объект (BLOB) или большой символьный объект (CLOB).

5.5.3. Синтаксис переходов в SQL

Синтаксис переходов предоставляет средства, которые обычно поддерживаются базами данных, но в разных вариантах в зависимости от конкретного синтаксиса базы данных. А в задачу драйвера JDBC входит преобразование синтаксиса переходов в синтаксис конкретной базы данных.

Переходы предусмотрены для следующих средств.

- Литералы времени и даты.
- Вызовы скалярных функций.
- Вызовы хранимых процедур.
- Внешние соединения.
- Символы перехода в операторах LIKE.

Литералы даты и времени сильно отличаются в разных базах данных. Чтобы вставить литерал даты или времени, нужно определить его значение в формате ISO 8601 (<http://www.cl.cam.ac.uk/~mgk25/iso-time.html>). После этого драйвер преобразует литерал в собственный формат базы данных. Для значений типа DATE, TIME или TIMESTAMP используются литералы d, t и ts следующим образом:

```
{d '2008-01-24'}
{t '23:59:59'}
{ts '2008-01-24 23:59:59.999'}
```

Скалярной называется такая функция, которая возвращает одно значение. В базах данных применяется немало скалярных функций, но под разными именами. В спецификации JDBC указаны стандартные имена, преобразуемые в имена, специфические для баз данных. Чтобы вызвать скалярную функцию, следует вставить ее стандартное имя и аргументы, как показано ниже. Полный список поддерживаемых имен скалярных функций можно найти в спецификации JDBC.

```
{fn left(?, 20)}
{fn user()}
```

Хранимой называется такая процедура, которая выполняется в базе данных и написана на языке, специфичном для конкретной базы данных. Для вызова хранимой процедуры служит переход call. Если у процедуры отсутствуют параметры, то указывать скобки не нужно. А для фиксации возвращаемого значения служит знак равенства. Ниже показано, каким образом вызываются хранимые процедуры.

```
{call PROC1(?, ?)}
{call PROC2}
{call ? = PROC3(?)}
```

Внешнее соединение двух таблиц не требует, чтобы строки из каждой таблицы совпадали по условию соединения. Например, в приведенном ниже запросе указаны книги, для которых столбец Publisher_Id не имеет совпадений в таблице Publishers, причем пустые значения NULL обозначают отсутствие совпадений.

```
SELECT * FROM {oj Books LEFT OUTER JOIN Publishers
    ON Books.Publisher_Id = Publisher.Publisher_Id}
```

Чтобы включить в запрос издательства без совпадающих книг, может потребоваться предложение RIGHT OUTER JOIN, а чтобы возвратить по запросу и то и другое — предложение FULL OUTER JOIN. Синтаксис переходов требуется именно потому, что не во всех базах данных используется стандартное обозначение внешних соединений.

И наконец, знаки _ и % имеют специальное значение в предложении LIKE, обозначая совпадение с одним символом или последовательностью символов. Стандартного способа их буквального использования не существует. Так, для соединения всех символьных строк, содержащих знак _, можно воспользоваться приведенной ниже конструкцией, где знак ! определен как символ перехода, а комбинация знаков !_ буквально обозначает знак подчеркивания.

```
... WHERE ? LIKE %!_% {escape '!'}
```

5.5.4. Множественные результаты

По запросу могут быть возвращены множественные результаты. Это может произойти при выполнении хранимой процедуры или в базах данных, которые допускают также выполнение многих команд SELECT в одном запросе. Получить все результирующие наборы можно следующим образом.

1. Вызвать метод execute() для выполнения команды SQL.
2. Получить первый результат или подсчет обновлений.
3. Повторить вызов метода getMoreResults(), чтобы перейти к следующему результирующему набору.
4. Завершить процедуру, если больше не остается результирующих наборов или подсчетов обновлений.

Методы execute() и getMoreResults() возвращают логическое значение true, если следующим звеном в цепочке оказывается результирующий набор. А метод getCount() возвращает значение -1, если следующим звеном в цепочке не оказывается подсчет обновлений. В следующем цикле осуществляется последовательный обход всех полученных результатов:

```
boolean isResult = stat.execute(command);
boolean done = false;
while (!done)
{
    if (isResult)
    {
        ResultSet result = stat.getResultSet();
        сделать что-нибудь с полученным результатом (result)
    }
    else
```

```

{
    int updateCount = stat.getUpdateCount();
    if (updateCount >= 0)
        сделать что-нибудь с подсчетом обновлений (updateCount)
    else
        done = true;
}
if (!done) isResult = stat.getMoreResults();
}

```

java.sql.Statement 1.1

- **boolean getMoreResults()**
- **boolean getMoreResults(int current)** 6

Получают следующий результат по данной команде SQL. Параметр **current** принимает значение одной из следующих констант: **CLOSE_CURRENT_RESULT** (по умолчанию), **KEEP_CURRENT_RESULT** или **CLOSE_ALL_RESULTS**. Возвращает логическое значение **true**, если следующий результат существует и представляет собой результирующий набор.

5.5.5. Извлечение автоматически генерируемых ключей

В большинстве баз данных поддерживается механизм автоматической нумерации строк в таблице. К сожалению, у разных поставщиков баз данных эти механизмы заметно отличаются. Автоматически присваиваемые номера часто используются в качестве первичных ключей. Несмотря на то что в JDBC не предлагается независимое от особенностей разных баз данных решение для генерирования подобных ключей, в этом прикладном интерфейсе предоставляется эффективный способ их извлечения. Если при вводе новой строки в таблицу автоматически генерируется ключ, его можно получить с помощью следующего кода:

```

stmt.executeUpdate(insertStatement, Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();
if (rs.next())
{
    int key = rs.getInt(1);
    ...
}

```

java.sql.Statement 1.1

- **boolean execute(String statement, int autogenerated)** 1.4
- **int executeUpdate(String statement, int autogenerated)** 1.4

Выполняют указанный оператор SQL, как пояснялось выше. Если параметр **autogenerated** принимает значение **Statement.RETURN_GENERATED_KEYS** и указана команда **INSERT**, то первый столбец таблицы содержит автоматически сгенерированный ключ.

5.6. Прокручиваемые и обновляемые результирующие наборы

Как пояснялось ранее, метод `next()` из интерфейса `ResultSet` позволяет последовательно перебирать строки в результирующем наборе, получаемом по запросу базы данных. Его очень удобно использовать для анализа полученных данных. Но нередко пользователю требуется предоставить возможность для просмотра результатов выполнения запроса с переходом к предыдущей и следующей строке, как было, например, показано на рис. 5.4. В прокручиваемом результирующем наборе можно свободно перемещаться не только к предыдущим и последующим записям, но и на произвольную позицию.

Кроме того, при просмотре результатов выполнения запроса у пользователей часто возникает потребность исправить какие-нибудь данные. В обновляемом результирующем наборе можно видоизменять записи программно, чтобы автоматически обновить их в базе данных. Все эти возможности обращения с результирующими наборами обсуждаются в последующих разделах.

5.6.1. Прокручиваемые результирующие наборы

По умолчанию результирующие наборы не являются прокручиваемыми или обновляемыми. Для организации прокрутки результатов выполнения запроса необходимо получить объект типа `Statement` следующим образом:

```
Statement stat = conn.createStatement(type, concurrency);
```

А для подготовленного оператора потребуется следующий вызов:

```
PreparedStatement stat =
    conn.prepareStatement(command, type, concurrency);
```

Допустимые значения параметров `type` и `concurrency` перечислены в табл. 5.6 и 5.7. Выбирая эти значения, придется найти ответы на следующие вопросы.

- Требуется ли сделать результирующий набор прокручиваемым? Если этого не требуется, следует выбрать значение `ResultSet.TYPE_FORWARD_ONLY`.
- Если все же требуется сделать результирующий набор прокручиваемым, то должен ли он отражать те данные, которые были изменены в базе данных после выполнения запроса? (Здесь и далее предполагается, что установлен параметр `ResultSet.TYPE_SCROLL_INSENSITIVE`, т.е. результирующий набор не “реагирует” на те изменения, которые произошли в базе данных после выполнения запроса.)
- Требуется ли отредактировать результирующий набор и обновить базу данных? (Более подробно этот вопрос рассматривается в следующем разделе.)

Так, если требуется только прокрутка результирующего набора, но не редактирование его данных, это можно организовать следующим образом:

```
Statement stat = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
```

Теперь можно прокручивать все результирующие наборы, возвращаемые при вызовах приведенного ниже метода. Получаемый в итоге результирующий набор содержит курсор, устанавливаемый на текущей позиции.

```
ResultSet rs = stat.executeQuery(query)
```

Таблица 5.6. Значения параметра type, представленные константами из интерфейса ResultSet

Значение	Описание
TYPE_FORWARD_ONLY	Без прокрутки (по умолчанию)
TYPE_SCROLL_INSENSITIVE	С прокруткой, но без учета изменений в базе данных
TYPE_SCROLL_SENSITIVE	С прокруткой и с учетом изменений в базе данных

Таблица 5.7. Значения параметра concurrency, представленные константами из интерфейса ResultSet

Значение	Описание
CONCUR_READ_ONLY	Без редактирования и обновления базы данных (по умолчанию)
CONCUR_UPDATABLE	С редактированием и обновлением базы данных



На заметку! Не все драйверы баз данных поддерживают прокручиваемые или обновляемые результирующие наборы. (Методы `supportsResultSetType()` и `supportsResultSetConcurrency()` из интерфейса `DatabaseMetaData` сообщают о типах и режимах параллельной обработки, которые поддерживаются в конкретной базе данных с помощью определенного драйвера.) Но даже если в базе данных поддерживаются результирующие наборы во всех описанных режимах, то в некоторых запросах нельзя получить результирующий набор со всеми запрашиваемыми свойствами. (Например, результат выполнения сложного запроса может оказаться необновляемым.)

В этом случае метод `executeQuery()` возвращает результирующий набор типа `ResultSet` с меньшими возможностями, вводя предупреждение типа `SQLWarning` в объект соединения. (Способ извлечения предупреждений представлен ранее, в разделе 5.4.3.) С другой стороны, для выявления конкретного режима работы результирующего набора можно вызвать методы `getType()` и `getConcurrency()` из интерфейса `ResultSet`. Если не выяснить конкретный режим работы результирующего набора и попытаться выполнить неподдерживаемую в нем операцию, например, вызвать метод `previous()` для непрокручиваемого результирующего набора, это неизбежно приведет к исключению типа `SQLException`.

Прокрутка организуется очень просто. Например, для перехода к предыдущим записям в результирующем наборе служит приведенная ниже конструкция. Метод `previous()` возвращает логическое значение `true`, если курсор находится на конкретной строке в результирующем наборе, и логическое значение `false`, если курсор находится перед первой строкой.

```
if (rs.previous()) . . .
```

Для перемещения курсора на *n* строк вперед или назад вызывается следующий метод:

```
rs.relative(n);
```

При положительных значениях параметра *n* курсор перемещается вперед, а при отрицательных — назад (нулевое значение параметра *n* не приводит ни к каким перемещениям). Если попытаться переместить курсор за пределы текущего ряда строк, он расположится за последней строкой или же перед первой строкой в зависимости от знака в значении параметра *n*. После этого метод `relative()` возвращает логическое значение `false`, а перемещение курсора

прекращается. Данный метод возвращает логическое значение `true` только в том случае, если курсор устанавливается на конкретной строке.

С другой стороны, курсор можно установить на конкретной строке под номером `п`, вызвав следующий метод:

```
s.absolute(п);
```

А получить текущий номер строки `п` можно следующим образом:

```
int currentRow = rs.getRow();
```

Первая строка в результирующем наборе имеет номер **1**. Если возвращаемое значение равно нулю, то курсор находится не на конкретной строке, а за последней или перед первой строкой. Для установки курсора на первой или последней строке, перед первой или за последней строкой результирующего набора предусмотрены удобные методы `first()`, `last()`, `beforeFirst()` и `afterLast()` соответственно, а для проверки расположения курсора на одной из этих позиций — удобные методы `isFirst()`, `isLast()`, `isBeforeFirst()` и `isAfterLast()`.

Как видите, обращаться с прокручиваемыми результирующими наборами совсем не трудно. Все рутинные операции, связанные с кешированием данных, получаемых по запросу, выполняются драйвером базы данных.

5.6.2. Обновляемые результирующие наборы

Если требуется отредактировать данные, полученные по запросу в результирующем наборе, а также автоматически обновить базу данных, такой набор нужно сделать обновляемым. Обновляемые результирующие наборы совсем не обязательно должны быть прокручиваемыми. Но если требуется предоставить пользователю возможность редактировать данные, то они должны допускать и прокрутку.

Для получения обновляемого результирующего набора служит приведенный ниже код, где для этой цели в качестве параметра указана константа, выделенная полужирным. Результирующие наборы, возвращаемые методом `executeQuery()`, становятся в итоге обновляемыми.

```
Statement stat = conn.createStatement()
ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
```



На заметку! Обновляемый результирующий набор возвращается не по всем запросам. Так, если в запросе предполагается соединение нескольких таблиц, его результат не всегда может быть обновляемым. Но если в запросе предполагается обращение к одной таблице или соединение нескольких таблиц по их первичным ключам, то следует ожидать, что получаемый в итоге результирующий набор окажется обновляемым. Чтобы выяснить, является ли результирующий набор обновляемым, следует вызвать метод `getConcurrency()` из интерфейса `ResultSet`.

Допустим, требуется повысить цену на некоторые книги, но отсутствует единый критерий, который можно было бы использовать для этого в команде `UPDATE`. В таком случае придется перебрать в цикле все книги и изменить цены по произвольным условиям, как показано ниже.

```
String query = "SELECT * FROM Books";
ResultSet rs = stat.executeQuery(query);
while (rs.next())
{
```

```

if (...)
{
    double increase = ...;
    double price = rs.getDouble("Price");
    rs.updateDouble("Price", price + increase);
    rs.updateRow(); // непременно вызвать метод updateRow()
                    // после обновления полей в таблице
}
}

```

Для всех типов данных SQL предусмотрены соответствующие формы метода updateXxx(), например updateDouble(), updateString() и т.д. Как и при вызове метода getXxx(), в качестве параметров данного метода могут быть указаны номер или имя столбца, а затем новое значение для поля.



На заметку! Применяя метод updateXxx(), следует иметь в виду, что первый его параметр обозначает номер столбца в результирующем наборе, где он может отличаться от номера столбца в базе данных.

Метод updateXxx() изменяет только значения полей в текущей строке результирующего набора, а не в самой базе данных. Для обновления всех полей отредактированной строки в базе данных следует вызвать метод updateRow(). Если же переместить курсор к следующей строке, не вызывая метод updateRow(), то все обновления предыдущей строки в результирующем наборе будут отменены, поскольку они не были переданы базе данных. Для отмены обновлений текущей строки в базе данных следует вызвать метод cancelRowUpdates().

В предыдущем примере был продемонстрирован порядок внесения изменений в существующей строке. Для создания новой строки в базе данных нужно сначала вызвать метод moveToInsertRow(), переместив тем самым курсор на специальную позицию, называемую *строкой вставки*. Затем новая строка создается на данной позиции с помощью метода updateXxx(). А для передачи новой вставляемой строки в базу данных вызывается метод insertRow(). По окончании вставки вызывается метод moveToCurrentRow(), чтобы переместить курсор назад на ту позицию, которую он занимал до вызова метода moveToInsertRow(). В приведенном ниже примере показано, каким образом весь этот процесс реализуется непосредственно в коде.

```

rs.moveToInsertRow();
rs.updateString("Title", title);
rs.updateString("ISBN", isbn);
rs.updateString("Publisher_Id", pubid);
rs.updateDouble("Price", price);
rs.insertRow();
rs.moveToCurrentRow();

```

Следует, однако, иметь в виду, что конкретное расположение новых данных в результирующем наборе или базе данных *не* поддается непосредственному управлению из прикладного кода. Если не указать конкретное значение для столбца в строке вставки, на этом месте окажется пустое значение NULL. Но если на столбец наложено ограничение NOT NULL, то генерируется исключение и строка не будет вставлена.

И наконец, для удаления той строки, на которой установлен курсор, вызывается метод, приведенный ниже. Он немедленно удаляет строку как из результирующего набора, так из базы данных.

```
rs.deleteRow();
```

Таким образом, методы `updateRow()`, `insertRow()` и `deleteRow()` из класса `ResultSet` предоставляют те же возможности, что и команды UPDATE, INSERT и DELETE языка SQL. Для программирующих на Java вызов методов более привычен, и поэтому они предпочитают данный подход составлению запросов из команд SQL.

Внимание! При неаккуратном обращении с обновляемыми результирующими наборами можно получить совершенно неэффективный код. Нередко выполнение команды `UPDATE` оказывается намного более эффективным, чем составление запроса и просмотр результирующего набора. Обработку обновляемых результирующих наборов имеет смысл организовывать в диалоговых прикладных программах, где пользователь может вносить произвольные изменения. А для внесения заранее программируемых изменений больше подходит команда `UPDATE`.

На заметку! В версии JDBC 2 были внедрены дополнительные усовершенствования в результирующие наборы, в том числе возможность обновлять результирующие наборы самыми последними данными, если они были изменены при другом, параллельном соединении с базой данных. А в версии JDBC 3 было внедрено еще одно усовершенствование, определяющее режим работы результирующих наборов при фиксации транзакции. Но эти дополнительные возможности здесь не рассматриваются, поскольку они выходят за рамки введения в базы данных. За дополнительными сведениями о них отсылаем читателей к книге *JDBC™ API Tutorial and Reference, Third Edition* Мэйдена Фишера, Джона Эллиса и Джонатана Брюса (Maydene Fisher, Jon Ellis, Jonathan Bruce; издательство Addison-Wesley, 2003 г.), а также к документации, описывающей спецификацию прикладного интерфейса JDBC и доступной для загрузки по адресу http://download.oracle.com/otndocs/jcp/jdbc-4_2-mrel2-spec/.

`java.sql.Connection 1.1`

- `Statement createStatement(int type, int concurrency) 1.2`
- `PreparedStatement prepareStatement(String command, int type, int concurrency) 1.2`

Создают обычный или подготовленный оператор SQL и возвращают результирующий набор.

Параметры:

`command`

Оператор, подготавливаемый
в виде команды SQL

`type`

Одна из констант
`(TYPE_FORWARD_ONLY,`
`CONCUR_UPDATABLE` или
`TYPE_SCROLL_SENSITIVE)`,
определяемых в интерфейсе

`ResultSet`

`concurrency`

Одна из констант
`(CONCUR_READ_ONLY` или
`CONCUR_UPDATABLE)`,
определяемых в интерфейсе

`ResultSet`

java.sql.ResultSet 1.1**• int getType() 1.2**

Возвращает одну из констант [**TYPE_FORWARD_ONLY**, **CONCUR_UPDATABLE** или **TYPE_SCROLL_SENSITIVE**], обозначающих тип результирующего набора.

• int getConcurrency() 1.2

Возвращает константу [**CONCUR_READ_ONLY** или **CONCUR_UPDATABLE**], обозначающую способ параллельного обращения к результирующему набору (только для чтения или обновления).

• boolean previous() 1.2

Перемещает курсор к предыдущей строке. Возвращает логическое значение **true**, если курсор устанавливается на строке, или логическое значение **false**, если он устанавливается перед первой строкой.

• int getRow() 1.2

Получает номер текущей строки. Нумерация строк начинается с 1.

• boolean absolute(int r) 1.2

Перемещает курсор к строке с номером **r**. Возвращает логическое значение **true**, если курсор устанавливается на строке.

• boolean relative(int d) 1.2

Перемещает курсор на количество строк, определяемое параметром **d**. Если значение параметра **d** меньше нуля, то перемещение происходит в обратном направлении. Возвращает логическое значение **true**, если курсор устанавливается на строке.

• boolean first() 1.2**• boolean last() 1.2**

Перемещают курсор к первой или к последней строке. Возвращают логическое значение **true**, если курсор устанавливается на строке.

• void beforeFirst() 1.2**• void afterLast() 1.2**

Устанавливают курсор перед первой или за последней строкой.

• boolean isFirst() 1.2**• boolean isLast() 1.2**

Проверяют, находится ли курсор на первой или на последней строке.

• boolean isBeforeFirst() 1.2**• boolean isAfterLast() 1.2**

Проверяют, находится ли курсор перед первой или за последней строкой.

• void moveToInsertRow() 1.2

Перемещает курсор на строку вставки. Стока вставки — это специальная строка, которая служит для вставки новых данных с помощью методов **updateXXX()** и **insertRow()**.

• void moveToCurrentRow() 1.2

Перемещает курсор из строки вставки на строку, где он находился до вызова метода **moveToInsertRow()**.

• void insertRow() 1.2

Вводит содержимое строки вставки в базу данных и результирующий набор.

java.sql.ResultSet 1.1 (окончание)

- **void deleteRow() 1.2**
Удаляет текущую строку из базы данных и результирующего набора.
- **void updateXxx(int column, Xxx data) 1.2**
- **void updateXxx(String columnName, Xxx data) 1.2**
(*Xxx* обозначает тип данных, например `int`, `double`, `String`, `Date` и т.д.)
Обновляют содержимое указанного столбца из текущей строки в результирующем наборе.
- **void updateRow() 1.2**
Передает обновления текущей строки в базу данных.
- **void cancelRowUpdates () 1.2**
Отменяет обновления текущей строки.

java.sql.DatabaseMetaData 1.1

- **boolean supportsResultSetType(int type) 1.2**

Возвращает логическое значение `true`, если база данных способна поддерживать заданный тип результирующего набора.

Параметры: **type** Одна из констант
`(TYPE_FORWARD_ONLY,`
`TYPE_SCROLL_INSENSITIVE` или
`TYPE_SCROLL_SENSITIVE),`
 определенных в интерфейсе
`ResultSet` и обозначающих способ прокрутки

- **boolean supportsResultSetConcurrency(int type, int concurrency) 1.2**

Возвращает логическое значение `true`, если база данных способна поддерживать заданный способ прокрутки и параллельного обращения к результирующему набору.

Параметры: **type** Одна из констант
`(TYPE_FORWARD_ONLY,`
`TYPE_SCROLL_INSENSITIVE` или
`TYPE_SCROLL_SENSITIVE),`
 определенных в интерфейсе `ResultSet`
 и обозначающих способ прокрутки
concurrency Одна из констант
`[CONCUR_READ_ONLY` или
`CONCUR_UPDATABLE],`
 определенных в интерфейсе `ResultSet`
 и обозначающих способ параллельного
 обращения к результирующему набору
 (только для чтения или обновления)

5.7. Наборы строк

Прокручиваемые результирующие наборы предлагают богатые возможности, но они не свободны от недостатков. В течение всего периода взаимодействия с пользователем должно быть установлено соединение с базой данных. Но ведь пользователь может отлучиться на длительное время, а между тем установленное соединение будет напрасно потреблять сетевые ресурсы. В подобной ситуации целесообразно использовать набор строк. Интерфейс RowSet расширяет интерфейс ResultSet, но набор строк не должен быть привязан к соединению с базой данных.

Наборы строк применяются и в том случае, если требуется перенести результаты выполнения запроса на другой уровень сложного приложения или на другое устройство, например, на мобильный телефон. Перенести результирующий набор нельзя, поскольку он связан с соединением, а кроме того, структура данных может иметь довольно крупные размеры.

5.7.1. Создание наборов строк

Ниже перечислены интерфейсы, входящие в пакет javax.sql.rowset и расширяющие интерфейс RowSet.

- Интерфейс CachedRowSet позволяет выполнять некоторые операции при отсутствии соединения. Кешируемые наборы строк рассматриваются в следующем разделе.
- Интерфейс WebRowSet представляет кешируемый набор строк, который может быть сохранен в XML-файле. А сам XML-файл может быть передан другому компоненту приложения и открыт с помощью другого объекта типа WebRowSet.
- Интерфейсы FilteredRowSet и JoinRowSet поддерживают легковесные операции с наборами строк, равнозначные таким командам SQL, как SELECT и JOIN. Эти операции выполняются только над данными, содержащимися в наборе строк, для чего не требуется устанавливать соединение с базой данных.
- Интерфейс JdbcRowSet является тонкой оболочкой для интерфейса ResultSet, вводя удобные методы из интерфейса RowSet.

В версии Java 7 появился стандартный способ получения набора строк с помощью приведенных ниже методов. Аналогичные методы имеются для получения наборов строк других типов.

```
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet crs = factory.createCachedRowSet();
```

До версии Java 7 применялись методы создания наборов строк, специфические для разных баз данных. Кроме того, в пакете com.sun.rowset, входящем в состав JDK, предоставляются базовые реализации, позволяющие применять наборы строк даже в том случае, если в конкретной базе данных они не поддерживаются. Имена классов в этих реализациях оканчиваются на Impl, например CachedRowSetImpl. Как показано ниже, к этим классам можно прибегнуть, если, например, нельзя воспользоваться классом RowSetProvider.

```
CachedRowSet crs = new com.sun.rowset.CachedRowSetImpl();
```

5.7.2. Кешируемые наборы строк

Кешируемый набор строк содержит данные из результирующего набора. Интерфейс CachedRowSet расширяет интерфейс ResultSet, а следовательно, кешируемым набором строк можно пользоваться точно так же, как и результирующим. Но наборы строк имеют существенное преимущество, позволяющее разорвать соединение с базой данных и продолжать работать с набором строк. Как демонстрируется в примере программы, исходный код которой будет представлен в листинге 5.4, такая возможность существенно упрощает создание диалоговых приложений. При получении команды от пользователя устанавливается соединение, выполняется запрос, результаты размещаются в наборе строк, после чего соединение с базой данных разрывается.

Кешируемый набор строк позволяет даже видоизменить содержащиеся в нем данные. Разумеется, результаты подобных изменений не отражаются в базе данных немедленно. Чтобы принять накопленные изменения, необходимо выполнить явный запрос. В этом случае объект типа CachedRowSet повторно устанавливает соединение и выдает команды SQL для записи изменений в базу данных. Объект типа CachedRowSet заполняется данными из результирующего набора следующим образом:

```
ResultSet result = . . .;
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet crs = factory.createCachedRowSet();
crs.populate(result);
conn.close(); // теперь можно разорвать соединение с базой данных
```

С другой стороны, объекту типа CachedRowSet можно предоставить возможность автоматически установить соединение. Для этого сначала задаются следующие параметры базы данных:

```
crs.setURL("jdbc:derby://localhost:1527/COREJAVA");
crs.setUsername("dbuser");
crs.setPassword("secret");
```

Затем составляется оператор запроса с любыми параметрами, как показано ниже.

```
crs.setCommand("SELECT * FROM Books WHERE PUBLISHER = ?");
crs.setString(1, publisherName);
```

И, наконец, набор строк заполняется результатами запроса. В результате приведенного ниже вызова устанавливается соединение с базой данных, выполняется запрос, заполняется набор строк, а затем соединение разрывается.

```
crs.execute();
```

Если полученный результат запроса имеет слишком большой объем, его можно и не полностью вводить в набор строк. В конце концов, пользователи прикладной программы, скорее всего, просмотрят лишь некоторые строки. В таком случае нужно определить размеры страницы следующим образом:

```
CachedRowSet crs = . . .;
crs.setCommand(command);
crs.setPageSize(20);
. . .
crs.execute();
```

В итоге будут доступны только **20** строк. А для того чтобы получить следующую порцию строк, достаточно вызвать метод

```
crs.nextPage();
```

Для просмотра и видоизменения набора строк служат те же методы, что и для обращения с результирующим набором. Так, если изменить содержимое набора строк, для записи изменений в базу данных необходимо сделать один из следующих вызовов:

```
crs.acceptChanges(conn);
```

или

```
crs.acceptChanges();
```

Второй вариант вызова метода `acceptChanges()` действует только в том случае, если предоставить для набора строк все сведения, необходимые для соединения с базой данных (URL, имя пользователя и пароль).

Как упоминалось в разделе 5.6.2, не все результирующие наборы являются обновляемыми. Аналогично наборы строк, содержащие результаты сложных запросов, не позволяют записывать изменения в базу данных. Если же набор строк содержит данные только из одной таблицы, то никаких затруднений при их записи в базу данных не возникает.



Внимание! Если заполнить набор строк данными из результирующего набора, то набору строк не будет известно имя обновляемой таблицы. В таком случае нужно специально указать имя таблицы, вызвав метод `setTableName()`.

Если данные в базе изменились с того момента, как набор строк был заполнен ими, то возникают дополнительные затруднения, связанные с несоответствием данных. В базовой реализации проверяется, совпадают ли исходные значения из набора строк (т.е. значения перед редактированием) с текущими значениями в базе данных. Если проверка дает положительный результат, то содержимое базы данных заменяется видоизмененными данными. В противном случае генерируется исключение типа `SyncProviderException` и внесенные изменения не записываются. В других реализациях могут применяться иные способы синхронизации данных.

javax.sql.RowSet 1.4

- `String getURL()`
- `void setURL(String url)`

Получают или устанавливают URL базы данных.

- `String getUsername()`
- `void setUsername(String username)`

Получают или устанавливают имя пользователя для соединения с базой данных.

- `String getPassword()`
- `void setPassword(String password)`

Получают или устанавливают пароль для соединения с базой данных.

javax.sql.RowSet 1.4 (окончание)

- **String getCommand()**
- **void setCommand(String command)**

Получают или устанавливают команду, при выполнении которой набор строк заполняется данными.

- **void execute()**

Заполняет данную строку по команде, установленной с помощью метода **setCommand()**. Для того чтобы диспетчер драйверов мог установить соединение, должны быть заданы URL, имя пользователя и пароль.

javax.sql.rowset.CachedRowSet 5.0

- **void execute(Connection conn)**

Заполняет набор строк по команде, установленной с помощью метода **setCommand()**. Использует указанное соединение с базой данных, а затем разрывает его.

- **void populate(ResultSet result)**

Заполняет кешируемый набор строк данными из указанного результирующего набора.

- **String getTableName()**

- **void setTableName(String tableName)**

Получают или устанавливают имя таблицы, данными из которой заполняется кешируемый набор строк.

- **int getPageSize()**

- **void setPageSize(int size)**

Получают или устанавливают размер страницы.

- **boolean nextPage()**

- **boolean previousPage()**

Загружают следующую или предыдущую страницу строк. Возвращают логическое значение **true**, если существует следующая или предыдущая страница.

- **void acceptChanges()**

- **void acceptChanges(Connection conn)**

Повторно устанавливают соединение с базой данных и записывают в нее изменения, внесенные в набор строк. Если с момента заполнения набора содержимое базы данных изменилось, данные не могут быть записаны в нее обратно. В этом случае генерируется исключение типа **SyncProviderException**.

javax.sql.rowset.RowSetProvider 7

- **static RowSetFactory newFactory()**

Создает фабрику наборов строк.

javax.sql.rowset.RowSetFactory 7

- **CachedRowSet createCachedRowSet()**
- **FilteredRowSet createFilteredRowSet()**
- **JdbcRowSet createJdbcRowSet()**
- **JoinRowSet createJoinRowSet()**
- **WebRowSet createWebRowSet()**

Создают набор строк заданного типа.

5.8. Метаданные

В предыдущих разделах рассматривались способы ввода и обновления содержимого таблиц базы данных и составления запросов. Помимо этого, в JDBC предусмотрены дополнительные возможности для получения сведений о структуре таблиц и самой базы данных. В частности, можно получить список всех таблиц базы данных или имена всех столбцов с типами данных в них. Эти сведения вряд ли будут очень полезны при разработке прикладной программы, предназначенной для работы с конкретной базой данных, потому что в таких случаях ее структура точно известна. Но они пригодятся и тем разработчикам, которые создают свои программные продукты для работы с любыми базами данных.

В языке SQL сведения о структуре базы данных и ее компонентов называются *метаданными*. Такое название выбрано лишь для того, чтобы как-то отличать сведения о базе данных от ее основного содержимого. Существуют метаданные трех типов, описывающие структуру базы данных, структуру результирующих наборов и параметры подготовленных операторов.

Для получения более подробных сведений о структуре базы данных требуется объект типа `DatabaseMetaData`, который можно получить из установленного соединения с базой данных следующим образом:

```
DatabaseMetaData meta = conn.getMetaData();
```

Далее можно приступать непосредственно к получению метаданных. Так, если вызвать приведенный ниже метод, то в итоге будет получен результирующий набор, содержащий сведения обо всех таблицах базы данных. (Параметры этого метода рассматриваются далее при описании соответствующего прикладного программного интерфейса API.)

```
ResultSet mrs =
    meta.getTables(null, null, null, new String[] { "TABLE" });
```

Каждая строка из получаемого в итоге результирующего набора содержит сведения об отдельной таблице, а третий столбец в ней — имя таблицы, как поясняется далее в описании соответствующего прикладного программного интерфейса API. В приведенном ниже фрагменте кода организуется цикл для сбора сведений об именах всех таблиц в базе данных.

```
while (mrs.next())
    tableName.addItem(mrs.getString(3));
```

Метаданные базы данных находят еще одно полезное применение. Базы данных могут иметь очень сложную структуру, а стандарт SQL предоставляет немало места для отклонений от нормы. Поэтому в интерфейсе DatabaseMetaData предусмотрено более сотни разных методов, которые можно использовать для получения сведений о структуре базы данных. Ниже приведены примеры вызова таких методов с довольно необычными именами. Судя по названиям этих методов, они предназначены главным образом для очень опытных разработчиков, в том числе тех, кто занимается написанием переносимого кода, способного работать с разнотипными базами данных.

```
meta.supportsCatalogsInPrivilegeDefinitions()  
и  
meta.nullPlusNonNullIsNull()
```

Интерфейс DatabaseMetaData предоставляет сведения о базе данных, а сведения о результирующем наборе — второй интерфейс ResultSetMetaData. Получив результирующий набор по запросу, можно определить количество столбцов, имена столбцов, типы данных в них и ширину полей. Ниже приведен типичный цикл, в котором все эти сведения извлекаются с помощью соответствующих методов, выделенных полужирным.

```
ResultSet rrs = stat.executeQuery("SELECT * FROM " + tableName);  
ResultSetMetaData meta = rrs.getMetaData();  
for (int i = 1; i <= meta.getColumnCount(); i++)  
{  
    String columnName = meta.getColumnLabel(i);  
    int columnWidth = meta.getColumnDisplaySize(i);  
    . . .  
}
```

В этом разделе поясняется, как создать простое инструментальное средство, предназначенное для просмотра и анализа структуры базы данных. Исходный код примера программы, реализующей это средство, приведен в листинге 5.4. В этой программе демонстрируется также применение кешированного набора строк.

В верхней части рабочего окна рассматриваемой здесь программы находится комбинированный список с именами всех таблиц базы данных. Как показано на рис. 5.6, после выбора какой-нибудь одной таблицы в центральной части фрейма будут представлены имена столбцов из этой таблицы, а также значения из первой строки. Для просмотра строк в таблице следует щелкнуть на кнопках Next (Следующая) и Previous (Предыдущая). Строки можно удалять, а также редактировать значения в них. Чтобы сохранить изменения в базе данных, следует щелкнуть на кнопке Save (Сохранить).



На заметку! В состав баз данных обычно включаются инструментальные средства для просмотра и редактирования таблиц, обладающие намного большими возможностями. Если для вашей базы такое инструментальное средство отсутствует, попробуйте воспользоваться iSQL-Viewer (<http://isql.sourceforge.net>) или SQuirreL (<http://squirrel-sql.sourceforge.net>). Эти инструментальные средства позволяют просматривать таблицы любой базы данных, совместимой с прикладным интерфейсом JDBC. Рассматриваемая здесь программа отнюдь не претендует на то, чтобы соперничать с этими инструментальными средствами. Она лишь демонстрирует общие принципы создания программ для работы с произвольными таблицами базы данных.

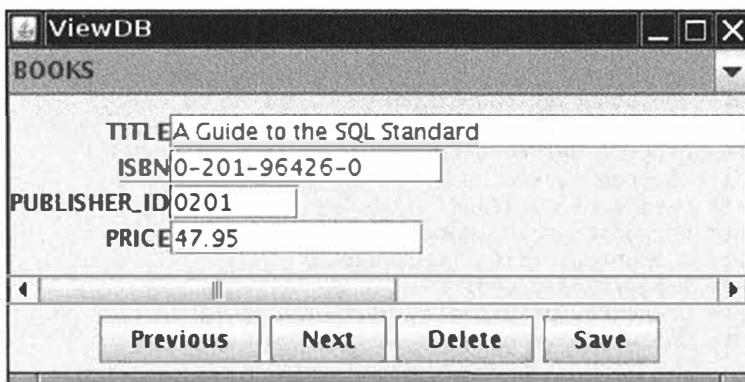


Рис. 5.6. Прикладная программа ViewDB

Листинг 5.4. Исходный код из файла view/ViewDB.java

```

1  package view;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.io.*;
6  import java.nio.file.*;
7  import java.sql.*;
8  import java.util.*;
9
10 import javax.sql.*;
11 import javax.sql.rowset.*;
12 import javax.swing.*;
13
14 /**
15 * В этой программе демонстрируется применение метаданных для
16 * отображения произвольно выбираемых таблиц в базе данных
17 * @version 1.33 2016-04-27
18 * @author Cay Horstmann
19 */
20 public class ViewDB
21 {
22     public static void main(String[] args)
23     {
24         EventQueue.invokeLater(() ->
25         {
26             JFrame frame = new ViewDBFrame();
27             frame.setTitle("ViewDB");
28             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29             frame.setVisible(true);
30         });
31     }
32 }
33
34 /**
35 * Фрейм, содержащий панель с кнопками перемещения по данным

```

```
36  */
37 class ViewDBFrame extends JFrame
38 {
39     private JButton previousButton;
40     private JButton nextButton;
41     private JButton deleteButton;
42     private JButton saveButton;
43     private DataPanel dataPanel;
44     private Component scrollPane;
45     private JComboBox<String> tableNames;
46     private Properties props;
47     private CachedRowSet crs;
48     private Connection conn;
49
50     public ViewDBFrame()
51     {
52         tableNames = new JComboBox<String>();
53
54         try
55         {
56             readDatabaseProperties();
57             conn = getConnection();
58             DatabaseMetaData meta = conn.getMetaData();
59             try (ResultSet mrs = meta.getTables(null, null, null,
60                     new String[] { "TABLE" })) {
61                 while (mrs.next())
62                     tableNames.addItem(mrs.getString(3));
63             }
64         }
65     }
66     catch (SQLException ex)
67     {
68         for (Throwable t : ex)
69             t.printStackTrace();
70     }
71     catch (IOException ex)
72     {
73         ex.printStackTrace();
74     }
75
76     tableNames.addActionListener(
77         event -> showTable(
78             (String) tableNames.getSelectedItem(), conn));
79     add(tableNames, BorderLayout.NORTH);
80     addWindowListener(new WindowAdapter()
81     {
82         public void windowClosing(WindowEvent event)
83         {
84             try
85             {
86                 if (conn != null) conn.close();
87             }
88             catch (SQLException ex)
89             {
90                 for (Throwable t : ex)
91                     t.printStackTrace();
92             }
93         }
94     });
95 }
```

```
92         }
93     }
94   });
95
96   JPanel buttonPanel = new JPanel();
97   add(buttonPanel, BorderLayout.SOUTH);
98
99   previousButton = new JButton("Previous");
100  previousButton.addActionListener(event -> showPreviousRow());
101  buttonPanel.add(previousButton);
102
103  nextButton = new JButton("Next");
104  nextButton.addActionListener(event -> showNextRow());
105  buttonPanel.add(nextButton);
106
107  deleteButton = new JButton("Delete");
108  deleteButton.addActionListener(event -> deleteRow());
109  buttonPanel.add(deleteButton);
110
111  saveButton = new JButton("Save");
112  saveButton.addActionListener(event -> saveChanges());
113  buttonPanel.add(saveButton);
114  if (tableNames.getItemCount() > 0)
115    showTable(tableNames.getItemAt(0), conn);
116 }
117
118 /**
119 * Подготавливает текстовые поля для показа новой таблицы и
120 * отображает первую ее строку
121 * @param tableName Имя отображаемой таблицы
122 * @param conn Соединение с базой данных
123 */
124 public void showTable(String tableName, Connection conn)
125 {
126   try (Statement stat = conn.createStatement();
127       ResultSet result = stat.executeQuery(
128           "SELECT * FROM " + tableName))
129   {
130     // получить результирующий набор
131
132     // скопировать его в кешируемый результирующий набор
133     RowSetFactory factory = RowSetProvider.newFactory();
134     crs = factory.createCachedRowSet();
135     crs.setTableName(tableName);
136     crs.populate(result);
137
138     if (scrollPane != null) remove(scrollPane);
139     dataPanel = new DataPanel(crs);
140     scrollPane = new JScrollPane(dataPanel);
141     add(scrollPane, BorderLayout.CENTER);
142     pack();
143     showNextRow();
144   }
145   catch (SQLException ex)
146   {
147     for (Throwable t : ex)
```

```
148         t.printStackTrace();
149     }
150 }
151
152 /**
153 * Осуществляет переход к предыдущей строке таблицы
154 */
155 public void showPreviousRow()
156 {
157     try
158     {
159         if (crs == null || crs.isFirst()) return;
160         crs.previous();
161         dataPanel.showRow(crs);
162     }
163     catch (SQLException ex)
164     {
165         for (Throwable t : ex)
166             t.printStackTrace();
167     }
168 }
169
170 /**
171 * Осуществляет переход к следующей строке таблицы
172 */
173 public void showNextRow()
174 {
175     try
176     {
177         if (crs == null || crs.isLast()) return;
178         crs.next();
179         dataPanel.showRow(crs);
180     }
181     catch (SQLException ex)
182     {
183         for (Throwable t : ex)
184             t.printStackTrace();
185     }
186 }
187
188 /**
189 * Удаляет строку из текущей таблицы
190 */
191 public void deleteRow()
192 {
193     if (crs == null) return;
194     new SwingWorker<Void, Void>()
195     {
196         public Void doInBackground() throws SQLException
197         {
198             crs.deleteRow();
199             crs.acceptChanges(conn);
200             if (crs.isAfterLast())
201                 if (!crs.last()) crs = null;
202             return null;
203         }
204     }
205 }
```

```
204     public void done()
205     {
206         dataPanel.showRow(crs);
207     }
208     }.execute();
209 }
210
211 /**
212 * Сохраняет все внесенные изменения
213 */
214 public void saveChanges()
215 {
216     if (crs == null) return;
217     new SwingWorker<Void, Void>()
218     {
219         public Void doInBackground() throws SQLException
220         {
221             dataPanel.setRow(crs);
222             crs.acceptChanges(conn);
223             return null;
224         }
225     }.execute();
226 }
227
228 private void readDatabaseProperties() throws IOException
229 {
230     props = new Properties();
231     try (InputStream in = Files.newInputStream(
232                         Paths.get("database.properties")))
233     {
234         props.load(in);
235     }
236     String drivers = props.getProperty("jdbc.drivers");
237     if (drivers != null)
238         System.setProperty("jdbc.drivers", drivers);
239 }
240
241 /**
242 * Получает соединение с базой данных из свойств,
243 * определенных в файле database.properties
244 * @return Возвращает соединение с базой данных
245 */
246 private Connection getConnection() throws SQLException
247 {
248     String url = props.getProperty("jdbc.url");
249     String username = props.getProperty("jdbc.username");
250     String password = props.getProperty("jdbc.password");
251
252     return DriverManager.getConnection(url, username, password);
253 }
254 }
255
256 /**
257 * Панель для отображения содержимого результирующего набора
258 */
259 class DataPanel extends JPanel
```

```
260 {
261     private java.util.List<JTextField> fields;
262
263     /**
264      * Конструирует панель для отображения данных
265      * @param rs Результирующий набор, содержимое которого
266      *          отображается на данной панели
267     */
268     public DataPanel(ResultSet rs) throws SQLException
269     {
270         fields = new ArrayList<>();
271         setLayout(new GridBagLayout());
272         GridBagConstraints gbc = new GridBagConstraints();
273         gbc.gridwidth = 1;
274         gbc.gridheight = 1;
275
276         ResultSetMetaData rsmd = rs.getMetaData();
277         for (int i = 1; i <= rsmd.getColumnCount(); i++)
278         {
279             gbc.gridx = i - 1;
280
281             String columnName = rsmd.getColumnName(i);
282             gbc.gridx = 0;
283             gbc.anchor = GridBagConstraints.EAST;
284             add(new JLabel(columnName), gbc);
285
286             int columnWidth = rsmd.getColumnDisplaySize(i);
287             JTextField tb = new JTextField(columnWidth);
288             if (!rsmd.getColumnClassName(i).equals("java.lang.String"))
289                 tb.setEditable(false);
290
291             fields.add(tb);
292
293             gbc.gridx = 1;
294             gbc.anchor = GridBagConstraints.WEST;
295             add(tb, gbc);
296         }
297     }
298
299     /**
300      * Отображает строку из таблицы базы данных, заполняя
301      * все текстовые значениями из столбцов
302     */
303     public void showRow(ResultSet rs)
304     {
305         try
306         {
307             if (rs == null) return;
308             for (int i = 1; i <= fields.size(); i++)
309             {
310                 String field = rs == null ? "" : rs.getString(i);
311                 JTextField tb = fields.get(i - 1);
312                 tb.setText(field);
313             }
314         }
315         catch (SQLException ex)
```

```

316     {
317         for (Throwable t : ex)
318             t.printStackTrace();
319     }
320 }
321
322 /**
323 * Обновляет измененными данными текущую строку
324 * из результирующего набора
325 */
326 public void setRow(ResultSet rs) throws SQLException
327 {
328     for (int i = 1; i <= fields.size(); i++)
329     {
330         String field = rs.getString(i);
331         JTextField tb = fields.get(i - 1);
332         if (!field.equals(tb.getText()))
333             rs.updateString(i, tb.getText());
334     }
335     rs.updateRow();
336 }
337 }
```

java.sql.Connection 1.1

- **DatabaseMetaData getMetaData()**

Возвращает метаданные в виде объекта типа **DatabaseMetaData** для соединения с базой данных.

java.sql.DatabaseMetaData 1.1

- **ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String types[])**

Возвращает из указанного каталога описание всех таблиц, совпадающих с шаблонами схемы и имен таблиц, а также с заданными критериями типов. (Схема описывает группу связанных вместе таблиц и полномочия доступа к ним, а каталог — группу связанных вместе схем. Эти понятия важны для структурирования крупных баз данных.)

В качестве параметров **catalog** и **schemaPattern** могут быть указаны пустые символьные строки (""), чтобы извлечь таблицы без каталога и схемы, или же пустые значения **null**, если требуется возвратить таблицы независимо от каталога или схемы.

Массив **types** содержит следующие имена типов таблиц: **TABLE**, **VIEW**, **SYSTEM TABLE**, **GLOBAL TEMPORARY**, **LOCAL TEMPORARY**, **ALIAS** и **SYNONYM**. Если вместо массива **types** указано пустое значение **null**, возвращаются таблицы всех типов.

Результирующий набор состоит из пяти столбцов типа **String**, как показано ниже.

Столбец	Имя	Описание
1	TABLE_CAT	Каталог таблиц (может иметь пустое значение null)

java.sql.DatabaseMetaData 1.1 (окончание)

2	TABLE_SCHEM	Схема (может иметь пустое значение null)
3	TABLE_NAME	Имя таблицы
4	TABLE_TYPE	Имя таблицы
5	REMARKS	Комментарии к таблице

- `int getJDBCMajorVersion() 1.4`
 - `int getJDBCMajorVersion() 1.4`

Возвращают основной и дополнительный номера версии драйвера JDBC, устанавливающего соединение с базой данных. Например, драйвер JDBC 3.0 имеет основной номер версии 3 и дополнительный номер версии 0.

- int getMaxConnections()

Возвращает максимальное количество соединений с базой данных, которые допускается устанавливать одновременно.

- int getMaxStatements()

Возвращает максимальное количество команд, которые допускается одновременно открывать на каждое соединение с базой данных. Если это количество неограничено или неизвестно, то возвращается нулевое значение.

java.sql.ResultSet 1.1

- `ResultSetMetaData getMetaData()`

Возвращает метаданные, связанные со столбцами текущего результирующего набора типа ResultSet

java.sql.ResultSetMetaData 1.1

- `int getColumnCount()`

Возвращает количество столбцов для текущего результирующего набора типа **ResultSet**.

- `int getColumnDisplaySize(int column)`

Возвращает максимальную ширину столбца по указанному целочисленному индексу

Параметры: **само** Номер столбца:

- ### **String getColumnLabel(int column)**

Возвращают предложенный заголовок для указанного столбца

Возвращает предполагаемый заголовок для указанного стиля.

- Страница `setColumnNamesList` возвращает

String getColumnName(int column)

Возвращает имя столбца по указанному целочисленному идентификатору.

5.9. Транзакции

Группа команд может быть оформлена в виде *транзакции*, которая может быть *зарегистрирована* после успешного выполнения всех команд или *откатана*, если при выполнении хотя бы одной из команд произойдет какая-нибудь ошибка. Основной причиной для группирования команд в транзакции служит сохранение целостности базы данных.

Допустим, требуется перевести денежные средства с одного банковского счета на другой. Для этого следует одновременно снять нужную сумму денег с одного счета и пополнить ею другой счет. Если после снятия суммы денег с одного счета, но перед пополнением другого произойдет системная ошибка, то операция с первым счетом должна быть отменена.

Команды обновления базы данных могут быть сгруппированы в одну транзакцию. Если транзакция завершается полностью, то возможна ее *фиксация*. А если она не завершается полностью из-за каких-нибудь ошибок, то производится ее *откат*, т.е. отменяются все изменения в базе данных, которые выполнялись после предыдущей зафиксированной транзакции.

5.9.1. Программирование транзакций средствами JDBC

По умолчанию соединение с базой данных находится в режиме *автоматической фиксации*, т.е. результат выполнения каждой команды SQL фиксируется в базе данных после успешного завершения этой команды. Как только команда будет зафиксирована, откатить ее уже нельзя. Для отключения режима автоматической фиксации можно вызвать следующий метод:

```
conn.setAutoCommit(false);
```

Отключив автоматическую фиксацию, можно приступать к созданию объекта типа *Statement* обычным образом:

```
Statement stat = conn.createStatement();
```

Затем метод *executeUpdate()* вызывается нужное количество раз, как показано ниже.

```
stat.executeUpdate(команда1);
stat.executeUpdate(команда2);
stat.executeUpdate(команда3);
...
```

Если все эти команды завершатся успешно, то результаты их выполнения фиксируются. Для этого вызывается метод *commit()* следующим образом:

```
conn.commit();
```

А если при выполнении любой из этих команд произойдет ошибка, то производится откат всей транзакции. И для этого вызывается метод *rollback()* следующим образом:

```
conn.rollback();
```

При этом автоматически отменяются все команды, выполнявшиеся после фиксации последней транзакции. Откат обычно производится в том случае, если при выполнении транзакции генерируется исключение типа `SQLException`.

5.9.2. Точки сохранения

Некоторые драйверы позволяют повысить уровень контроля над процессом отката с помощью *точек сохранения*. При создании точки сохранения отмечается точка, в которую можно впоследствии вернуться, не отменяя всю транзакцию. В приведенном ниже фрагменте кода показано, как это осуществляется на практике.

```
Statement stat = conn.createStatement(); // начать транзакцию;
// переход в эту точку происходит при вызове метода rollback()
stat.executeUpdate(команда1);
Savepoint svpt = conn.setSavepoint(); // установить точку сохранения;
// переход в эту точку происходит при вызове метода rollback(svpt)
stat.executeUpdate(команда2);
if ( . . . ) conn.rollback(svpt); // отменить результат выполнения
// команды2
.
.
.
conn.commit();
```

Если точка сохранения больше не нужна, ее следует освободить следующим образом:

```
conn.releaseSavepoint(svpt);
```

5.9.3. Групповые обновления

Допустим, в программе требуется выполнить много команд `INSERT` для заполнения таблицы базы данных. Повысить производительность такой программы можно с помощью *группового обновления*. При групповом обновлении команды собираются вместе и выдаются группой, а не по отдельности.



На заметку! Чтобы выяснить, поддерживается ли в базе данных групповое обновление, достаточно вызвать метод `supportsBatchUpdates()` из интерфейса `DatabaseMetaData`.

Помимо команд управления данными `INSERT`, `UPDATE` и `DELETE`, для группового обновления можно также использовать команды определения данных, в том числе `CREATE TABLE` и `DROP TABLE`. Но для этой цели не подходит команда `SELECT`, поскольку ее выполнение в группе с другими командами приводит к исключению. (В принципе не имеет смысла выполнять команду `SELECT` в групповом режиме, поскольку она возвращает результирующий набор, не обновляя базу данных.)

Для группового обновления сначала создается объект типа `Statement`:

```
Statement stat = conn.createStatement();
```

Затем вместо метода `executeUpdate()` вызывается метод `addBatch()`:

```
String command = "CREATE TABLE . . . "
stat.addBatch(команда);
while ( . . . )
```

```
{
    command = "INSERT INTO . . . VALUES (" + . . . + ")";
    stat.addBatch(команда);
}
```

И наконец, все команды выдаются вместе для группового обновления базы данных, как показано ниже. Метод `executeBatch()` возвращает массив подсчетов строк, обработанных при выполнении каждой команды из данной группы.

```
int[] counts = stat.executeBatch();
```

Для правильной обработки ошибок в групповом режиме групповое обновление следует рассматривать как единую транзакцию. Если в ходе группового обновления произойдет сбой или возникнет ошибки, следует произвести откат в исходное состояние.

Прежде всего следует отключить режим автоматической фиксации, собрать команды в группу, выполнить их и зафиксировать результаты, а затем восстановить режим автоматической фиксации, как выделено полужирным в приведенном ниже фрагменте кода.

```
boolean autoCommit = conn.getAutoCommit();
conn.setAutoCommit(false);
Statement stat = conn.createStatement();
. . .
// продолжать вызовы метода stat.addBatch(. . .);
. . .
stat.executeBatch();
conn.commit();
conn.setAutoCommit(autoCommit);
```

`java.sql.Connection 1.1`

- `boolean getAutoCommit()`
- `void setAutoCommit(boolean b)`

Получают или устанавливают режим автоматической фиксации для данного соединения с базой данных. Если параметр `b` принимает логическое значение `true`, результаты выполнения всех команд автоматически фиксируются после их завершения.

- `void commit()`
- Фиксирует все команды, которые были выданы с момента последней фиксации.

- `void rollback()`
- Производит откат, отменяя все изменения, которые были внесены с момента последней фиксации.

- `Savepoint setSavepoint() 1.4`

Устанавливают безымянную или именованную точку сохранения.

- `void rollback(Savepoint svpt) 1.4`

Производит откат всех команд до указанной точки сохранения.

- `void releaseSavepoint(Savepoint svpt) 1.4`

Освобождает указанную точку сохранения.

java.sql.Savepoint 1.4

- **int getSavepointId()**

Возвращает идентификатор данной безымянной точки сохранения. Если данная точка оказывается именованной, генерируется исключение типа **SQLException**.

- **String getSavepointName()**

Возвращает имя точки сохранения. Если данная точка оказывается безымянной, генерируется исключение типа **SQLException**.

java.sql.Statement 1.1

- **void addBatch(String command) 1.2**

Включает указанную команду в текущую группу для группового обновления.

- **int[] executeBatch() 1.2**

- **long[] executeLargeBatch() 8**

Выполняют все команды из текущей группы. Каждое значение в возвращаемом массиве соответствует одной из команд, входящих в данную группу. Если это положительное значение, то оно обозначает подсчет обновленных строк. Если же это значение **SUCCESS_NO_INFO**, то оно обозначает, что команду удалось выполнить, но подсчет обновленных строк недоступен. А если это значение **EXECUTE_FAILED**, то оно обозначает, что выполнить команду не удалось.

java.sql.DatabaseMetaData 1.1

- **boolean supportsBatchUpdates() 1.2**

Возвращает логическое значение **true**, если драйвер поддерживает групповое обновление.

5.10. Расширенные типы данных SQL

В табл. 5.8 перечислены все типы данных SQL, поддерживаемых в JDBC, а также их эквиваленты в Java.

Таблица 5.8. Типы данных SQL и соответствующие им типы в Java

Тип данных SQL	Тип данных Java
INTEGER или INT	int
SMALLINT	short
NUMERIC (m, n), DECIMAL (m, n) или DEC (m, n)	java.math.BigDecimal
FLOAT (n)	double
REAL	float
DOUBLE	double
CHARACTER (n) или CHAR (n)	String
VARCHAR (n), LONG VARCHAR	String
BOOLEAN	boolean
DATE	java.sql.Date

Окончание табл. 5.8

Тип данных SQL	Тип данных Java
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
BLOB	java.sql.Blob
CLOB	java.sql.Clob
ARRAY	java.sql.Array
ROWID	java.sql.RowId
NCHAR (n), NVARCHAR (n), LONG NVARCHAR	String
NCLOB	java.sql.NClob
SQLXML	java.sql.SQLXML

Тип ARRAY представляет в SQL последовательность значений. Например, таблица Student может иметь столбец с оценками Scores типа ARRAY OF INTEGER, т.е. с массивом целочисленных значений. А метод `getArray()` возвращает объект интерфейса типа `java.sql.Array`. В этом интерфейсе предусмотрены также методы извлечения значений из массива.

При получении большого объекта (LOB) или массива из базы данных конкретное содержимое извлекается из нее только после запроса отдельных значений. Это сделано для повышения эффективности работы с базой данных, поскольку большой объект может оказаться довольно массивным.

В некоторых базах данных поддерживаются значения типа ROWID, описывающие местонахождение строки, что позволяет очень быстро извлечь ее из таблицы. В версии JDBC 4 внедрен интерфейс `java.sql.RowId`, предоставляющий методы для ввода идентификатора строки в запросы и извлечения его из получаемых результатов.

Строка национальных символов (типа NCHAR и его вариантов) служит для хранения символьных строк в локальной кодировке, а также для их сортировки по заданным условиям локальной сортировки. В версии JDBC 4 предоставляются методы для взаимного преобразования объектов Java типа `String` и строк национальных символов в запросах и получаемых результатах.

В некоторых базах данных допускается хранение данных, типы которых определяются пользователем. В версии JDBC 3 поддерживается механизм автоматического преобразования структурированных типов данных SQL в объекты Java. Кроме того, в некоторых базах данных предоставляется собственный механизм хранения данных в формате XML. В версии JDBC 4 внедрен интерфейс `SQLXML`, который может служить связующим звеном между внутренним представлением данных в формате XML и интерфейсами `Source/Result` модели DOM, а также двоичными потоками ввода-вывода. Дополнительные сведения об интерфейсе `SQLXML` можно найти в документации на соответствующий прикладной программный интерфейс API.

На этом рассмотрение расширенных типов данных SQL в этой главе завершается. Дополнительные сведения по этому вопросу можно найти в упоминавшейся ранее книге *JDBC™ API Tutorial and Reference, Third Edition* и спецификации прикладного интерфейса JDBC 4.

5.11. Управление подключением к базам данных в веб-приложениях и производственных приложениях

Описанный ранее способ соединения с базой данных с помощью параметров из файла свойств `database.properties` подходит только для очень простых тестовых программ и совершенно не годится для крупномасштабных приложений. При установке приложения JDBC в корпоративной среде соединения с базами данных поддерживаются через интерфейс JNDI (Java Naming and Directory Interface — интерфейс для служб каталогов и именования в Java). Свойства источников данных в пределах всего предприятия хранятся в отдельном каталоге. Благодаря этому обеспечивается централизованное управление именами пользователей, паролями и URL в JDBC. В такой среде для установления соединения с базой данных рекомендуется использовать код, подобный следующему:

```
Context jndiContext = new InitialContext();
DataSource source =
    (DataSource) jndiContext.lookup("java:comp/env/jdbc/corejava");
Connection conn = source.getConnection();
```

Обратите внимание на то, что в этом коде уже не используется диспетчер драйверов типа `DriverManager`. Вместо него для поиска источника данных применяется служба JNDI. В качестве источника данных служит интерфейс `DataSource`, позволяющий устанавливать простые соединения типа JDBC, а также выполнять ряд более сложных функций, например, распределенные транзакции с несколькими базами данных. Интерфейс `DataSource` входит в пакет `javax.sql`, расширяющий стандартную библиотеку Java.



На заметку! В контейнере Java EE не нужно даже программировать поиск в службе JNDI. Достаточно сделать следующую аннотацию `Resource` к полю типа `DataSource`, чтобы во время загрузки приложения был автоматически задан эталонный источник данных:

```
@Resource(name="jdbc/corejava")
private DataSource source;
```

Очевидно, что источник данных нуждается в настройке. Так, если прикладная программа для работы с базой данных должна выполняться в контейнере серверов (например, Apache Tomcat) или на сервере приложений (например, GlassFish), то сведения о настройке базы данных (в том числе имя службы JNDI, URL в JDBC, имя пользователя и пароль) целесообразно разместить в конфигурационном файле или же задать в ГПИ администратора.

Управление именами пользователей и регистрационными данными — это лишь один из вопросов, требующих особого внимания. Второй вопрос связан со стоимостью установления соединений с базами данных. В примерах программ, представленных в этой главе, применяются две методики для получения требующегося соединения с базой данных. Так, в самом начале программы `QueryDB` из листинга 5.3 устанавливается единственное соединение с базой данных, которое разрывается по завершении программы. А в программе `ViewDB` из листинга 5.4 новое соединение устанавливается всякий раз, когда в нем возникает потребность.

Но ни одну из этих методик нельзя считать удовлетворительной. Ведь соединения с базами данных — это конечный ресурс. И если пользователь приостановит работу с приложением на некоторое время, то соединение не следует оставлять установленным. А с другой стороны, получение соединения для каждого запроса и последующий его разрыв обходится очень дорого.

В качестве выхода из этого положения целесообразно организовать пул соединений. Это означает, что соединения с базами данных не разрываются физически, а сохраняются в очереди и повторно используются для других запросов. Организация пулов соединений является важной служебной функцией, и поэтому в спецификации JDBC предоставляются вспомогательные средства для ее реализации. Следует, однако, иметь в виду, что в различных базах данных пул соединений может быть реализован по-разному. Причем он не всегда входит в состав драйвера JDBC. Некоторые поставщики веб-контейнеров и серверов приложений предлагают реализации пулов соединений в составе своих приложений.

Использование пула соединений полностью прозрачно для программиста. Чтобы извлечь соединение из пула, достаточно получить соответствующий источник данных и вызвать метод `getConnection()`. А по окончании работы с базой данных через это соединение следует вызвать метод `close()`. При этом соединение физически не разрывается, но в пуле соединений поступает сообщение о том, что оно больше не требуется. Как правило, в пуле соединений принимаются необходимые меры к тому, чтобы сохранить в нем и подготовленные операторы.

Итак, вы ознакомились с самыми основами JDBC, которые требуются для разработки простых прикладных программ, взаимодействующих с базами данных. Но, как отмечалось в начале этой главы, базы данных могут иметь очень сложную структуру, а более сложные вопросы работы с ними выходят за рамки данной книги. Поэтому интересующихся подобными вопросами еще раз отсылаем к книге *JDBC™ API Tutorial and Reference, Third Edition* и спецификации JDBC.

Из этой главы вы узнали о том, каким образом организуется взаимодействие с реляционными базами данных в Java. А следующая глава посвящена библиотеке даты и времени, внедренной в версии Java 8.

Прикладной программный интерфейс API даты и времени

В этой главе...

- ▶ Временная шкала
- ▶ Местные даты
- ▶ Корректоры дат
- ▶ Местное время
- ▶ Поясное время
- ▶ Форматирование и синтаксический анализ даты и времени
- ▶ Взаимодействие с унаследованным кодом

Время летит как стрела, и мы можем легко установить начальный момент, чтобы отсчитывать время вперед и назад. Так почему же так трудно обращаться со временем? Все дело в самих людях. Не проще ли было, если бы мы обращались друг к другу следующим образом: “Встретимся в 1371409200, только не опаздывай!” Но ведь нам нужно соотносить время с конкретным временем суток и года. Именно здесь и возникают трудности. В версии Java 1.0 имелся класс `Date`, реализация которого теперь кажется наивной, а большинство его методов стали нерекомендованными к употреблению, начиная с версии Java 1.1, где был внедрен класс `Calendar`. Безусловно, его прикладной программный интерфейс API не был совершенным, его экземпляры были изменяемыми, и в нем не учитывались

потерянные секунды. Более совершенным оказался прикладной программный интерфейс API даты и времени, внедренный в версии Java 8. В нем были устранены недостатки прежних реализаций, и можно надеяться, что он послужит нам еще немало времени. В этой главе будет показано, что именно делает расчеты времени столь неприятными и как подобные трудности разрешаются в прикладном программном интерфейсе API даты и времени.

6.1. Временная шкала

По традиции, основополагающей единицей отсчета времени является секунда, производная от вращения Земли вокруг своей оси. Полный оборот Земля совершает за 24 часа, или $24 \times 60 \times 60 = 86400$ секунд, поэтому точное определение секунды кажется делом астрономических измерений. К сожалению, Земля испытывает незначительные колебания при вращении, что потребовало более точного определения секунды. И такое определение было сформулировано в 1967 году. Оно вполне согласуется с исторически сложившимся определением и в то же время основывается на внутреннем свойстве атомов Цезия-133. С тех пор официальное время хранится в разветвленной сети атомных часов.

Нередко официальные часы-хранители времени синхронизируют абсолютное время с вращением Земли. Прежде официальные секунды подвергались незначительной коррекции, но с 1972 года стали периодически вводиться так называемые "потерянные" секунды. (Теоретически секунду можно было бы время от времени удалять, но этого так и не произошло.) В настоящее время снова ведутся дискуссии об изменении системы отсчета времени. Очевидно, что причиной тому служат потерянные секунды, и поэтому во многих вычислительных системах применяется так называемое "стгаживание" там, где время искусственно замедляется или ускоряется перед потерянной секундой, чтобы сохранить ровно 86400 секунд в сутках. Такой способ вполне работоспособен, поскольку местное время на компьютере отсчитывается не совсем точно, а компьютеры обычно синхронизируются с внешней службой времени.

В соответствии со спецификацией на прикладной программный интерфейс API для даты и времени в Java требуется временная шкала, которая

- имеет 86 400 секунд в сутках;
- точно соответствует официальному времени в полдень каждого дня;
- близко соответствует ему в другое время суток строго определенным способом.

Благодаря этому язык Java может гибко подстраиваться к будущим изменениям в отсчете официального времени. В языке Java класс `Instant` представляет точку на временной шкале. Исходная точка отсчета времени, называемая эпохой, произвольно задана в полночь 1 января 1970 года на нулевом меридиане, проходящем через Гринвичскую королевскую обсерваторию в Лондоне. Аналогичное соглашение принято и для отсчета времени по стандарту POSIX в системе Unix. Начиная с исходной точки отсчета, время измеряется в секундах, как вперед, так и назад, с точностью до наносекунд, а каждые сутки составляют 86400

секунд. При отсчете времени назад значения типа `Instant` достигают миллиарда лет (т.е. минимальной точки `Instant.MIN` на временной шкале). И хотя этого недостаточно, чтобы выразить возраст Вселенной (около 13,5 млрд лет), но должно быть достаточно для практических целей. Ведь миллиард лет назад Земля была покрыта льдом и населялась микроскопическими предшественниками современных растений и животных. А максимальная точка на временной шкале (`Instant.MAX`) соответствует 31 декабря 1000000000 года.

В результате вызова статического метода `Instant.now()` получается текущий момент времени. Два момента времени можно сравнить с помощью методов `equals()` и `compareTo()` обычным образом, чтобы использовать моменты времени как отметки времени.

Для определения разности двух моментов времени служит статический метод `Duration.between()`. В качестве примера ниже показано, как измерить текущее время выполнения алгоритма.

```
Instant start = Instant.now();
runAlgorithm();
Instant end = Instant.now();
Duration timeElapsed = Duration.between(start, end);
long millis = timeElapsed.toMillis();
```

Объект типа `Duration` определяет промежуток между двумя моментами времени. Длительность промежутка типа `Duration` в обычных единицах измерения времени можно получить, вызвав метод `toNanos()`, `toMillis()`, `toSeconds()`, `toMinutes()`, `toHours()` или `toDays()`.

Для внутреннего хранения промежутков времени требуется не только значение типа `long`. В частности, количество секунд хранится в значении типа `long`, а количество наносекунд — в дополнительном значении типа `int`. Так, если требуется произвести расчеты времени с точностью до наносекунд во всем промежутке типа `Duration`, то можно воспользоваться одним из методов, перечисленных в табл. 6.1. В противном случае достаточно вызвать метод `toNanos()`, чтобы произвести расчеты времени со значениями типа `long`.



На заметку! Чтобы переполнилось значение типа `long`, потребуется количество наносекунд порядка 300 лет.

Таблица 6.1. Методы, выполняющие арифметические операции над моментами и промежутками времени

Метод	Описание
<code>plus(), minus()</code>	Добавляет или вычитает промежуток времени из данного момента времени типа <code>Instant</code> или промежутка времени типа <code>Duration</code>
<code>plusNanos(), plusMillis(), plusSeconds(), plusMinutes(), plusHours(), plusDays()</code>	Добавляют количество заданных единиц измерения времени к данному моменту времени типа <code>Instant</code> или промежутку времени типа <code>Duration</code>
<code>minusNanos(), minusMillis(), minusSeconds(), minusMinutes(), minusHours(), minusDays()</code>	Вычитают количество заданных единиц измерения времени из данного момента времени типа <code>Instant</code> или промежутка времени типа <code>Duration</code>

Окончание табл. 6.1

Метод	Описание
<code>multipliedBy()</code> , <code>dividedBy()</code> , <code>negated()</code>	Возвращают промежуток времени, получаемый умножением или делением данного промежутка времени типа <code>Duration</code> на заданное значение типа <code>long</code> или <code>-1</code> . Масштабировать можно только промежутки, но не моменты времени
<code>isZero()</code> , <code>isNegative()</code>	Проверяют, является ли данный промежуток времени типа <code>Duration</code> нулевым или отрицательным

Так, если требуется проверить, выполняется ли один алгоритм, по крайней мере, в десять раз быстрее, чем другой, достаточно выполнить следующие расчеты:

```
Duration timeElapsed2 = Duration.between(start2, end2);
boolean overTenTimesFaster =
    timeElapsed.multipliedBy(10).minus(timeElapsed2).isNegative();
// или timeElapsed.toNanos() * 10 < timeElapsed2.toNanos()
```



На заметку! Классы `Instant` и `Duration` неизменяемы, и поэтому все методы вроде `multipliedBy()` или `minus()` возвращают новые экземпляры этих классов.

В примере программы из листинга 6.1 демонстрируется применение классов `Instant` и `Duration` для расчета времени выполнения двух алгоритмов.

Листинг 6.1. Исходный код из файла timeline/TimeLine.java

```
1 package timeline;
2
3 import java.time.*;
4 import java.util.*;
5 import java.util.stream.*;
6
7 public class Timeline
8 {
9     public static void main(String[] args)
10    {
11         Instant start = Instant.now();
12         runAlgorithm();
13         Instant end = Instant.now();
14         Duration timeElapsed = Duration.between(start, end);
15         long millis = timeElapsed.toMillis();
16         System.out.printf("%d milliseconds\n", millis);
17
18         Instant start2 = Instant.now();
19         runAlgorithm2();
20         Instant end2 = Instant.now();
21         Duration timeElapsed2 = Duration.between(start2, end2);
22         System.out.printf("%d milliseconds\n",
23                           timeElapsed2.toMillis());
24         boolean overTenTimesFaster = timeElapsed.multipliedBy(10)
25             .minus(timeElapsed2).isNegative();
26         System.out.printf(
27             "The first algorithm is %smore than ten times faster",
28             overTenTimesFaster ? "" : "not ");
```

```
29 }
30
31 public static void runAlgorithm()
32 {
33     int size = 10;
34     List<Integer> list =
35         new Random().ints().map(i -> i % 100).limit(size)
36         .boxed().collect(Collectors.toList());
37     Collections.sort(list);
38     System.out.println(list);
39 }
40
41 public static void runAlgorithm2()
42 {
43     int size = 10;
44     List<Integer> list =
45         new Random().ints().map(i -> i % 100).limit(size)
46         .boxed().collect(Collectors.toList());
47     while (!IntStream.range(1, list.size()).allMatch(
48         i -> list.get(i - 1).compareTo(list.get(i)) <= 0))
49         Collections.shuffle(list);
50     System.out.println(list);
51 }
52 }
```

6.2. Местные даты

А теперь перейдем от абсолютного к обычному в обиходе времени. Такое время в прикладном программном интерфейсе Java API представлено двумя категориями: *местного времени и даты и поясного времени*. Местное время и дата обозначают время суток или дату, но не связаны с часовым поясом. Примером местной даты служит 14 июня 1903 года (в этот день родился Алонсо Черч, изобретатель лямбда-вычислений). Эта дата не содержит ни время суток, ни часовой пояс и поэтому не соответствует точному моменту времени. С другой стороны, дата 16 июля 1969 года, 09:32:00 по восточному поясному времени (момент запуска космического корабля “Аполлон-11”) представляет точный момент времени на временной шкале с учетом часового пояса.

Во многих расчетах времени учитывать часовые пояса не требуется, а иногда они могут даже мешать. Допустим, требуется запланировать еженедельные совещания в 10:00. Если добавить 7 дней (т.е. $7 \times 24 \times 60 \times 60$ секунд) к последнему часовому поясу, то невольно можно пересечь границу, обозначающую переход на летнее или зимнее время, и тогда совещание состоится на час раньше или позже!

Именно по этой причине разработчики прикладного программного интерфейса API даты и времени рекомендуют не пользоваться поясным временем, кроме тех случаев, когда требуется представить экземпляры абсолютного времени. Дни рождения, праздники, сроки исполнения и прочие моменты времени лучше всего представить в виде местного времени и даты.

Объект типа `LocalDate` определяет местную дату с указанием года, месяца и дня месяца. Для создания этого объекта можно воспользоваться статическим методом `now()` или `of()`:

```
LocalDate today = LocalDate.now(); // Текущая дата
LocalDate alonzosBirthday = LocalDate.of(1903, 6, 14);
alonzoBirthday = LocalDate.of(1903, Month.JUNE, 14);
// Применяется перечисление Month
```

В отличие от нестандартных соглашений, принятых в системе Unix и классе `java.util.Date`, где отсчет месяцев начинается с нуля, а отсчет лет — с 1900 года, месяц года можно обозначить обычными числами. С другой стороны, для этой цели можно воспользоваться перечислением `Month`. В табл. 6.2 перечислены наиболее употребительные методы для обращения с объектами типа `LocalDate`.

Таблица 6.2. Методы из класса `LocalDate`

Метод	Описание
<code>now(), of()</code>	Эти статические методы служат для создания объекта типа <code>LocalDate</code> , исходя из текущего времени или заданного года, месяца и дня
<code>plusDays(), plusWeeks(), plusMonths(), plusYears()</code>	Добавляют количество дней, недель, месяцев или лет к местной дате, представленной объектом типа <code>LocalDate</code>
<code>minusDays(), minusWeeks(), minusMonths(), minusYears()</code>	Вычитают количество дней, недель, месяцев или лет из местной даты, представленной объектом типа <code>LocalDate</code>
<code>plus(), minus()</code>	Добавляют или вычитают промежуток времени типа <code>Duration</code> или период времени типа <code>Period</code>
<code>withDayOfMonth(), withDayOfYear(), withMonth(), withYear()</code>	Возвращают новую местную дату в виде объекта типа <code>LocalDate</code> с днем месяца, днем года или года, измененного на заданное значение
<code>getDayOfMonth()</code>	Получает день месяца в пределах от 1 до 31
<code>getDayOfYear()</code>	Получает день года в пределах от 1 до 366
<code>getDayOfWeek()</code>	Получает день недели, возвращая значение из перечисления <code>DayOfWeek</code>
<code>getMonth(), getMonthValue()</code>	Получают месяц в виде значения из перечисления <code>Month</code> или числа в пределах от 1 до 12
<code>getYear()</code>	Получает год в пределах от <code>-999999999</code> до <code>999999999</code>
<code>Until()</code>	Получает период времени типа <code>Period</code> или количество заданных единиц времени типа <code>ChronoUnit</code> в промежутке между двумя датами
<code>isBefore(), isAfter()</code>	Сравнивает данную местную дату типа <code>LocalDate</code> с другой датой
<code>isLeapYear()</code>	Возвращает логическое значение <code>true</code> , если год оказывается високосным, т.е. если он делится на 4, но не на 100 или 400. Этот алгоритм применяется ко всем предыдущим годам, хотя он исторически неточный. (Високосные годы были введены в 46 году до н.э., а правила деления на 100 или 400 — при реформе григорианского календаря в 1582 году. На повсеместное распространение этой реформы понадобилось более 300 лет.)

Например, День программиста приходится на 256-й день года. Ниже показано, насколько просто рассчитывается этот день.

```
LocalDate programmersDay = LocalDate.of(2014, 1, 1).plusDays(255);
// 13 сентября, но в високосный год — 12 сентября
```

Напомним, что разность двух моментов времени составляет промежуток типа Duration. Для местных дат ему соответствует период времени типа Period, который выражается количеством прошедших лет, месяцев или дней. Разумеется, для получения местной даты дня своего рождения в следующем году можно сделать вызов birthday.plus(Period.ofYears(1)). Но в високосный год в результате вызова birthday.plus(Duration.ofDays(365)) вряд ли будет получен правильный результат.

Метод until() возвращает разность двух местных дат. Например, в результате следующего вызова:

```
independenceDay.until(christmas)
```

получается период времени в 5 месяцев и 21 день, что не очень удобно, поскольку количество дней в месяце варьируется. Поэтому для выявления количества дней лучше сделать следующий вызов:

```
independenceDay.until(christmas, ChronoUnit.DAYS) // 174 дня
```



Внимание! Вызов некоторых методов из табл. 6.2 мог бы привести к получению несуществующих дат. Так, если добавить один месяц к дате 31 января, то в конечном итоге не должна получиться дата 31 февраля. Вместо генерирования исключения эти методы возвращают последний достоверный день месяца. Например, в результате одного из следующих вызовов:

```
LocalDate.of(2016, 1, 31).plusMonths(1)
```

или

```
LocalDate.of(2016, 3, 31).minusMonths(1)
```

получается дата 29 февраля 2016.

Метод getDayOfWeek() возвращает день недели в виде соответствующего значения из перечисления DayOfWeek. В частности, понедельнику соответствует значение DayOfWeek.MONDAY, равное 1, а воскресенью — значение DayOfWeek.SUNDAY, равное 7. Например, в результате следующего вызова:

```
LocalDate.of(1900, 1, 1).getDayOfWeek().getValue()
```

возвращается значение 1. У перечисления DayOfWeek имеются удобные методы plus() и minus() для расчета дней недели по модулю 7. Так, в результате вызова DayOfWeek.SATURDAY.plus(3) возвращается значение DayOfWeek.TUESDAY.



На заметку! Выходные дни фактически приходятся на конец недели. В то же время в классе java.util.Calendar воскресенью соответствует значение 1, а субботе — значение 7.

Помимо класса LocalDate, имеются классы MonthDay, YearMonth и Year для описания частичных дат. Например, дата 25 декабря (с указанным годом) может быть представлена объектом класса MonthDay. Применение класса LocalDate демонстрируется в примере программы из листинга 6.2.

Листинг 6.2. Исходный код из файла localdates/LocalDates.java

```
1 package localdates;
2
3 import java.time.*;
```

```
4 import java.time.temporal.*;
5
6 public class LocalDates
7 {
8     public static void main(String[] args)
9     {
10         LocalDate today = LocalDate.now(); // Текущая дата
11         System.out.println("today: " + today);
12
13         LocalDate alonzosBirthday = LocalDate.of(1903, 6, 14);
14         alonzosBirthday = LocalDate.of(1903, Month.JUNE, 14);
15         // Применяется перечисление Month
16         System.out.println("alonzosBirthday: " + alonzosBirthday);
17
18         LocalDate programmersDay =
19             LocalDate.of(2018, 1, 1).plusDays(255);
20         // 13 сентября, но в високосный год 12 сентября
21         System.out.println("programmersDay: " + programmersDay);
22
23         LocalDate independenceDay = LocalDate.of(2018, Month.JULY, 4);
24         LocalDate christmas = LocalDate.of(2018, Month.DECEMBER, 25);
25
26         System.out.println("Until christmas: "
27             + independenceDay.until(christmas));
28         System.out.println("Until christmas: "
29             + independenceDay.until(christmas, ChronoUnit.DAYS));
30
31         System.out.println(LocalDate.of(2016, 1, 31).plusMonths(1));
32         System.out.println(LocalDate.of(2016, 3, 31).minusMonths(1));
33
34         DayOfWeek startOfLastMillennium =
35             LocalDate.of(1900, 1, 1).getDayOfWeek();
36         System.out.println("startOfLastMillennium: "
37             + startOfLastMillennium);
38         System.out.println(startOfLastMillennium.getValue());
39         System.out.println(DayOfWeek.SATURDAY.plus(3));
40     }
41 }
```

6.3. Корректоры дат

Для целей планирования нередко требуется рассчитать такие даты, как первый вторник каждого месяца. В классе TemporalAdjusters предоставляется целый ряд статических методов для общих видов коррекции дат. Результат выполнения метода коррекции дат передается методу `with()`. Например, первый вторник месяца может быть рассчитан следующим образом:

```
LocalDate firstTuesday = LocalDate.of(year, month, 1).with(
    TemporalAdjusters.nextOrSame(DayOfWeek.TUESDAY));
```

Как всегда, метод `with()` возвращает новый объект типа `LocalDate`, не изменяя оригинал. Доступные корректоры дат перечислены табл. 6.3.

Таблица 6.3. Доступные корректоры дат из класса TemporalAdjusters

Метод	Описание
<code>next(weekday)</code> , <code>previous(weekday)</code>	Возвращают следующую или предыдущую дату, приходящуюся на день недели, определяемый параметром <code>weekday</code>
<code>nextOrSame(weekday)</code> , <code>previousOrSame(weekday)</code>	Возвращают следующую или предыдущую дату, приходящуюся на день недели, определяемый параметром <code>weekday</code> , начиная с указанной даты
<code>dayOfWeekInMonth(n, weekday)</code>	Возвращает <i>n</i> -й день недели в месяце, определяемый параметром <code>weekday</code>
<code>lastInMonth(weekday)</code>	Возвращает последний день недели в месяце, определяемый параметром <code>weekday</code>
<code>firstDayOfMonth()</code> , <code>firstDayOfNextMonth()</code> , <code>firstDayOfNextYear()</code> , <code>lastDayOfMonth()</code> , <code>lastDayOfPreviousMonth()</code> , <code>lastDayOfYear()</code>	Возвращают дату, обозначаемую в имени вызываемого метода

Имеется также возможность создать свой корректор дат, реализовав интерфейс `TemporalAdjuster`. В качестве примера ниже приведен корректор дат, предназначенный для расчета следующего дня недели.

```
TemporalAdjuster NEXT_WORKDAY = w -> {
    LocalDate result = (LocalDate) w;
    do {
        result = result.plusDays(1);
    } while (result.getDayOfWeek().getValue() >= 6);
    return result;
};
```

```
LocalDate backToWork = today.with(NEXT_WORKDAY);
```

Обратите внимание на то, что параметр лямбда-выражения относится к типу `Temporal`, и поэтому он должен быть приведен к типу `LocalDate`. Избежать этого приведения типов можно с помощью метода `ofDateAdjuster()`, ожидающего в качестве параметра лямбда-выражение типа `UnaryOperator<LocalDate>`:

```
TemporalAdjuster NEXT_WORKDAY = TemporalAdjusters.ofDateAdjuster(w -> {
    LocalDate result = w; // Без приведения типов
    do {
        result = result.plusDays(1);
    } while (result.getDayOfWeek().getValue() >= 6);
    return result;
});
```

6.4. Местное время

Класс `LocalTime` представляет местное время суток, например 15:30:00. Экземпляр класса `LocalTime` можно получить с помощью метода `now()` или `of()` следующим образом:

```
LocalTime rightNow = LocalTime.now();
LocalTime bedtime = LocalTime.of(22, 30);
// или LocalTime.of(22, 30, 0)
```

В табл. 6.4 перечислены методы, выполняющие общие операции с местным временем. В частности, методы `plus()` и `minus()` заключают в себе операции с местным временем в течение всех суток, как показано в следующей строке кода:

```
LocalTime wakeup = bedtime.plusHours(8); // Подъем в 6:30:00!
```

Таблица 6.4. Методы из класса `LocalTime`

Метод	Описание
<code>now(), of()</code>	Эти статические методы создают объект типа <code>LocalTime</code> , представляющий местное время, исходя из текущего времени или из заданных часов, минут, а возможно, секунд и наносекунд
<code>plusHours(), plusMinutes(), plusSeconds(), plusNanos()</code>	Добавляют количество часов, минут, секунд или наносекунд к местному времени, представленному объектом типа <code>LocalTime</code>
<code>minusHours(), minusMinutes(), minusSeconds(), minusNanos()</code>	Вычитают количество часов, минут, секунд или наносекунд из местного времени, представленного объектом типа <code>LocalTime</code>
<code>plus(), minus()</code>	Добавляют или вычтут промежуток времени типа <code>Duration</code>
<code>withHour(), withMinute(), withSecond(), withNano()</code>	Возвращают новый объект типа <code>LocalTime</code> , представляющий местное время, где час, минута, секунда или наносекунда изменены на заданное значение
<code>getHour(), getMinute(), getSecond(), getNano()</code>	Получают час, минуту, секунду или наносекунду местного времени, представленного объектом типа <code>LocalTime</code>
<code>toSecondOfDay(), toNanoOfDay()</code>	Возвращают количество секунд или наносекунд в промежутке между полуночью и местным временем, представленным объектом типа <code>LocalTime</code>
<code>isBefore(), isAfter()</code>	Сравнивают данное местное время, представленное объектом типа <code>LocalTime</code> , с другим местным временем



На заметку! В самом классе `LocalTime` местное время до и после полудня не разграничивается, поскольку эта обязанность возлагается на средство форматирования, как поясняется далее, в разделе 6.6.

Имеется также класс `LocalDateTime`, представляющий дату и время, который пригоден для хранения моментов времени в фиксированном часовом поясе, например, для планирования занятий или событий. Но если требуется произвести расчеты с учетом перехода на летнее время или поддерживать пользователей из разных часовых поясов, то для этих целей следует воспользоваться классом `ZonedDateTime`, рассматриваемым в следующем разделе.

6.5. Поясное время

Еще большую путаницу, чем неравномерность вращения Земли, в расчеты времени вносят часовые пояса, вероятно, потому, что они являются изобретением человечества. Люди во всем мире ведут отсчет от времени по Гринвичу, и поэтому одни обедают в 2:00, а другие в 22:00, исходя из потребностей своего желудка, а не времени суток. Так, в Китае обедают в разное время, поскольку на эту страну приходятся четыре условных часовых пояса. Эти пояса считаются условными, поскольку их границы неточны и смещаются, а переход на летнее и зимнее время еще больше усугубляет положение.

Как бы часовые пояса ни досаждали просвещенным, они являются непреложным фактом нашей жизни. Разрабатывая календарное приложение, нужно учитывать интересы людей, перемещающихся из одной страны в другую. Так, если требуется вовремя прибыть к 10:00 на конференцию в Нью-Йорке из Берлина, нужно правильно учесть местное время, определяя время отъезда.

В организации IANA (Internet Assigned Numbers Authority — Комитет по цифровым адресам в Интернете) ведется база данных всех известных в мире часовых поясов (<https://www.iana.org/time-zones>), обновляемая несколько раз в год. Большая часть обновлений относится к изменению правил перехода на летнее и зимнее время. База данных IANA применяется и в Java.

Каждому часовому поясу присваивается свой идентификатор, например America/New_York или Europe/Berlin. Чтобы выяснить все имеющиеся часовые пояса, достаточно вызвать метод ZoneId.getAvailableIds(). На момент написания данной книги насчитывалось более 600 идентификаторов часовых поясов.

Получая в качестве параметра идентификатор часового пояса, статический метод ZoneId.of(id) возвращает объект типа ZoneId. Этот объект можно использовать для преобразования объекта типа LocalDateTime в объект типа ZonedDateTime, вызвав метод local.atZone(zoneId), или для создания объекта типа ZonedDateTime, вызвав статический метод ZonedDateTime.of(year, month, day, hour, minute, second, nano, zoneId), как показано в следующем примере кода:

```
ZonedDateTime apollo11launch = ZonedDateTime.of(1969, 7, 16, 9, 32, 0, 0,  
        ZoneId.of("America/New_York"));  
// 1969-07-16T09:32-04:00[America/New_York]
```

Это конкретный момент времени. Чтобы получить объект типа Instant, достаточно сделать вызов apollo11launch.toInstant(). С другой стороны, если имеется конкретный момент времени, достаточно сделать вызов instant.atZone(ZoneId.of("UTC")), чтобы получить объект типа ZonedDateTime, определяющий поясное время и дату по Гринвичу, или воспользоваться другим объектом типа ZoneId, чтобы получить дату и время в любом другом месте планеты.



На заметку! Сокращение UTC обозначает Всеобщее скоординированное время. Это сокращение выбрано в качестве компромисса между английским (Coordinated Universal Time) и французским (Temps Universel Coordonné) обозначениями Всеобщего скоординированного времени, хотя оно неверно на обоих языках. Понятие UTC определяет время по Гринвичу без учета перехода на летнее или зимнее время.

Многие методы из класса ZonedDateTime похожи на методы из класса LocalDateTime (табл. 6.5). Большинство из них довольно просты, но переход на летнее и зимнее время несколько усложняет расчет поясного времени.

При переходе на летнее время стрелки часов переводятся на один час вперед. Что же произойдет, если рассчитать время, приходящееся на пропущенный час? Например, в 2013 году переход на летнее время в Центральной Европе произошел 31 марта в 2:00. Если попытаться рассчитать время в несуществующий момент 2:30 31 марта 2013 года, то на самом деле будет получено время 3:30:

```
ZonedDateTime skipped = ZonedDateTime.of(  
        LocalDate.of(2013, 3, 31),
```

```
LocalTime.of(2, 30),
ZoneId.of("Europe/Berlin"));
// Получается время 3:30 31 марта
```

С другой стороны, при переходе на зимнее время стрелки часов переводятся на один час назад, и в одно и то же местное время возникают два момента! При расчете времени в этом промежутке получается более ранний из этих двух моментов времени, как показано ниже.

```
ZonedDateTime ambiguous = ZonedDateTime.of(
    LocalDate.of(2013, 10, 27), // Переход на зимнее время
    LocalTime.of(2, 30),
    ZoneId.of("Europe/Berlin"));
// 2013-10-27T02:30+02:00[Europe/Berlin]
ZonedDateTime anHourLater = ambiguous.plusHours(1);
// 2013-10-27T02:30+01:00[Europe/Berlin]
```

Час спустя время будет показывать те же самые часы и минуты, но часовой пояс уже будет смещен. Следует также уделить внимание коррекции даты при пересечении границ перехода на летнее и зимнее время. Так, если требуется назначить совещание на следующей неделе, то не нужно вводить промежуток в семь дней, как в следующем примере кода:

```
ZonedDateTime nextMeeting = meeting.plus(Duration.ofDays(7));
// Внимание! Этот код неверен при переходе на летнее время
```

Вместо этого лучше воспользоваться классом `Period` следующим образом:

```
ZonedDateTime nextMeeting = meeting.plus(Period.ofDays(7)); // Верно!
```

Таблица 6.5. Методы из класса `ZonedDateTime`

Метод	Описание
<code>now(), of(), ofInstant()</code>	Эти статические методы строят объект типа <code>ZonedDateTime</code> , представляющий поясное время и дату, исходя из текущего времени или года, месяца, дня, часа, минуты, секунды, наносекунды или местного времени (объекта типа <code>LocalDate</code>) и даты (объекта типа <code>LocalTime</code>), а также идентификатора часового пояса (объекта типа <code>ZoneId</code>) или текущего момента времени (объекта типа <code>Instant</code>) и идентификатора часового пояса (объекта типа <code>ZoneId</code>)
<code>plusDays(), plusWeeks(), plusMonths(), plusYears(), plusHours(), plusMinutes(), plusSeconds(), plusNanos()</code>	Добавляют количество соответствующих единиц измерения времени к поясному времени и дате, представленных объектом типа <code>ZonedDateTime</code>
<code>minusDays(), minusWeeks(), minusMonths(), minusYears(), minusHours(), minusMinutes(), minusSeconds(), minusNanos()</code>	Вычитают количество соответствующих единиц измерения времени из поясного времени и даты, представленных объектом типа <code>ZonedDateTime</code>
<code>plus(), minus()</code>	Добавляют или вычтут промежуток времени типа <code>Duration</code> или период времени типа <code>Period</code>
<code>withDayOfMonth(), withDayOfYear(), withMonth(), withYear(), withHour(), withMinute(), withSecond(), withNano()</code>	Возвращают новый объект типа <code>ZonedDateTime</code> , представляющий поясное время и дату, где час, минута, секунда или наносекунда изменены на заданное значение

Окончание табл. 6.5

Метод	Описание
<code>withZoneSameInstant()</code> , <code>withZoneSameLocal()</code>	Возвращают новый объект типа <code>ZonedDateTime</code> , представляющий в заданном часовом поясе тот же самый момент времени или то же самое локальное время
<code>getDayOfMonth()</code>	Получает день месяца в пределах от 1 до 31
<code>getDayOfYear()</code>	Получает день года от 1 до 366
<code>getDayOfWeek()</code>	Получает день недели, возвращая значение из перечисления <code>DayOfWeek</code>
<code>getMonth(), getMonthValue()</code>	Получают месяц в виде значения из перечисления <code>Month</code> или числа в пределах от 1 до 12
<code>getYear()</code>	Получает год в пределах от <code>-999999999</code> до <code>999999999</code>
<code>getHour(), getMinute(), getSecond(), getNano()</code>	Получают час, минуту, секунду или наносекунду поясного времени, представленного объектом типа <code>ZonedDateTime</code>
<code>getOffset()</code>	Получает смещение относительно времени UTC в виде экземпляра типа <code>ZoneOffset</code> . Смещение может изменяться в пределах от <code>-12:00</code> до <code>+14:00</code> . У некоторых часовых поясов может быть дробное смещение. При переходе на летнее или зимнее время смещение изменяется
<code>toLocalDate(), toLocalTime(), toInstant()</code>	Получают локальную дату, локальное время или соответствующий момент времени
<code>isBefore(), isAfter()</code>	Сравнивает данное поясное время и дату типа <code>ZonedDateTime</code> с другим поясным временем и датой

Внимание! Имеется также класс `OffsetDateTime`, представляющий время со смещением относительно времени UTC, но без учета правил смены часовых поясов. Этот класс предназначен для специального применения, требующего, в частности, отсутствия этих правил, в том числе и некоторых сетевых протоколов. Для получения поясного времени в удобочитаемом формате лучше пользоваться классом `ZonedDateTime`.

Применение класса `ZonedDateTime` демонстрируется в примере программы из листинга 6.3.

Листинг 6.3. Исходный код из файла zonedtimes/ZonedDateTime.java

```

1 package zonedtimes;
2
3 import java.time.*;
4
5 public class ZonedDateTime
6 {
7     public static void main(String[] args)
8     {
9         ZonedDateTime apollo11lauch = ZonedDateTime.of(
10             1969, 7, 16, 9, 32, 0, 0, ZoneId.of("America/New_York"));
11         // 1969-07-16T09:32-04:00[America/New_York]
12         System.out.println("apollo11lauch: " + apollo11lauch);
13
14         Instant instant = apollo11lauch.toInstant();
15         System.out.println("instant: " + instant);
16     }
17

```

```

18     ZonedDateTime zonedDateTime = instant.atZone(ZoneId.of("UTC"));
19     System.out.println("zonedDateTime: " + zonedDateTime);
20
21     ZonedDateTime skipped = ZonedDateTime.of(
22         LocalDate.of(2013, 3, 31),
23         LocalTime.of(2, 30), ZoneId.of("Europe/Berlin"));
24     // Формирование даты 31 марта 3:30
25     System.out.println("skipped: " + skipped);
26
27     ZonedDateTime ambiguous = ZonedDateTime.of(
28         LocalDate.of(2013, 10, 27),
29         LocalTime.of(2, 30), ZoneId.of("Europe/Berlin"));
30     // Переход на зимнее время
31     // 2013-10-27T02:30+02:00[Europe/Berlin]
32     ZonedDateTime anHourLater = ambiguous.plusHours(1);
33     // 2013-10-27T02:30+01:00[Europe/Berlin]
34     System.out.println("ambiguous: " + ambiguous);
35     System.out.println("anHourLater: " + anHourLater);
36
37     ZonedDateTime meeting = ZonedDateTime.of(
38         LocalDate.of(2013, 10, 31),
39         LocalTime.of(14, 30), ZoneId.of("America/Los_Angeles"));
40     System.out.println("meeting: " + meeting);
41     ZonedDateTime nextMeeting = meeting.plus(Duration.ofDays(7));
42     // Внимание! Не годится, так как не учитывает
43     // переход с летнего времени на зимнее
44     System.out.println("nextMeeting: " + nextMeeting);
45     nextMeeting = meeting.plus(Period.ofDays(7)); // Верно!
46     System.out.println("nextMeeting: " + nextMeeting);
47 }
48 }
```

6.6. Форматирование и синтаксический анализ даты и времени

В классе `DateTimeFormatter` предоставляются следующие три вида средств форматирования для вывода значений даты и времени.

- Предопределенные стандартные средства форматирования (табл. 6.6).
- Средства форматирования с учетом региональных настроек.
- Средства форматирования по специальным шаблонам.

Чтобы воспользоваться одним из стандартных средств форматирования, достаточно вызвать его метод `format()` следующим образом:

```
String formatted = DateTimeFormatter.ISO_DATE_TIME.format(apollo11Launch);
// 1969-07-16T09:32:00-05:00[America/New_York]
```

Таблица 6.6. Предопределенные средства форматирования

Средство форматирования	Описание	Пример
<code>BASIC_ISO_DATE</code>	Год, месяц, день, смещение часового пояса без разделителей	19690716-0500
<code>ISO_LOCAL_DATE, ISO_LOCAL_TIME, ISO_LOCAL_DATE_TIME</code>	Разделители -, :, T	1969-07-16, 09:32:00, 1969-07-16T09:32:00

Окончание табл. 6.6

Средство форматирования	Описание	Пример
<code>ISO_OFFSET_DATE</code> , <code>ISO_OFFSET_TIME</code> , <code>ISO_OFFSET_DATE_TIME</code>	Аналогично <code>ISO_LOCAL_XXX</code> , но со смещением часового пояса	1969-07-16-05:00, 09:32:00-05:00, 1969-07-16T09:32:00-05:00
<code>ISO_ZONED_DATE_TIME</code>	Со смещением часового пояса и идентификатором часового пояса	1969-07-16T09:32:00-05:00 [America/New_York]
<code>ISO_INSTANT</code>	Время в формате UTC, где <code>z</code> обозначает идентификатор часового пояса	1969-07-16T14:32:00Z
<code>ISO_DATE</code> , <code>ISO_TIME</code> , <code>ISO_DATE_TIME</code>	Аналогично <code>ISO_OFFSET_DATE</code> , <code>ISO_OFFSET_TIME</code> и <code>ISO_ZONED_DATE_TIME</code> , но сведения о часовом поясе указываются дополнительно, хотя и не обязательно	1969-07-16-05:00, 09:32:00-05:00, 1969-07-16T09:32:00-05:00 [America/New_York]
<code>ISO_ORDINAL_DATE</code>	Год и день года для местной даты типа <code>LocalDate</code>	1969-197
<code>ISO_WEEK_DATE</code>	Год, неделя и день недели для местной даты типа <code>LocalDate</code>	1969-W29-3
<code>RFC_1123_DATE_TIME</code>	Стандарт для отметок времени в электронной почте, кодируемых по стандарту RFC 822 и обновляемых четырьмя цифрами для обозначения года по стандарту RFC 1123	Wed, 16 Jul 1969 09:32:00 -0500

Стандартные средства предназначены в основном для форматирования машинно-читаемых отметок времени. А для представления дат и времени в удобочитаемом виде служат средства форматирования с учетом региональных настроек. Они поддерживают четыре стиля форматирования даты и времени: `SHORT`, `MEDIUM`, `LONG` и `FULL` (табл. 6.7).

Таблица 6.7. Стили форматирования с учетом региональных настроек

Стиль	Дата	Время
<code>SHORT</code>	7/16/69	9:32 AM
<code>MEDIUM</code>	Jul 16, 1969	9:32:00 AM
<code>LONG</code>	July 16, 1969	9:32:00 AM EDT
<code>FULL</code>	Wednesday, July 16, 1969	9:32:00 AM EDT

Для создания средства форматирования с учетом региональных настроек служат статические методы `ofLocalizedDate()`, `ofLocalizedTime()` и `ofLocalizedDateTime()`. Ниже приведен характерный тому пример.

```
DateTimeFormatter formatter =
    DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG);
String formatted = formatter.format(apolloLaunch);
// July 16, 1969 9:32:00 AM EDT
```

В этих методах применяются региональные настройки, выбираемые по умолчанию. Чтобы выбрать другие региональные настройки, достаточно вызвать метод `withLocale()` следующим образом:

```
formatted = formatter.withLocale(Locale.FRENCH).format(apollo11Launch);
// 16 juillet 1969 09:32:00 EDT
```

У перечислений `DayOfWeek` и `Month` имеются методы `getDisplayName()` для получения наименований дней недели и месяца в разных форматах и региональных настройках, как показано ниже. Подробнее о региональных настройках речь пойдет в главе 7.

```
for (DayOfWeek w : DayOfWeek.values())
    System.out.print(
        w.getDisplayName(TextStyle.SHORT, Locale.ENGLISH) + " ");
    // Выводит дни недели Mon Tue Wed Thu Fri Sat Sun
    // на английском языке
```



На заметку! Класс `java.time.format.DateTimeFormatter` служит для замены класса `java.util.DateFormat`. Если же требуется получить экземпляр последнего ради обратной совместимости, следует вызвать метод `formatter.toFormat()`.

И наконец, можно создать свой формат даты, указав шаблон. Так, в следующей строке кода:

```
formatter = DateTimeFormatter.ofPattern("E yyyy-MM-dd HH:mm");
```

дата форматируется в виде `Wed 1969-07-16 09:32`. Каждая буква в шаблоне обозначает отдельное поле даты и времени, а количество повторений букв — конкретный формат, выбираемый по правилам, которые не совсем ясны и, по-видимому, органично выработались со временем. Наиболее употребительные элементы шаблонов для форматирования даты и времени перечислены в табл. 6.8.

Таблица 6.8. Наиболее употребительные знаки в шаблонах для форматирования даты и времени

Константа перечислимого типа <code>ChronoField</code> или назначение	Перевод	Примеры
<code>ERA</code>	Эра	<code>G: AD, GGGG: Anno Domini, GGGGG: A</code>
<code>YEAR_OF_ERA</code>	Год эры	<code>yy: 69, yyyy: 1969</code>
<code>MONTH_OF_YEAR</code>	Месяц года	<code>M: 7, MM: 07, MMM: Jul, MMMM: July, MMMMM: J</code>
<code>DAY_OF_MONTH</code>	День месяца	<code>d: 6, dd: 06</code>
<code>DAY_OF_WEEK</code>	День недели	<code>e: 3, E: Wed, EEEE: Wednesday, EEEEE: W</code>
<code>HOUR_OF_DAY</code>	Час дня	<code>H: 9, HH: 09</code>
<code>CLOCK_HOUR_OF_AM_PM</code>	Полный час дня по часам до или после полудня	<code>K: 9, KK: 09</code>
<code>AMPM_OF_DAY</code>	Время суток до или после полудня	<code>a: AM</code>
<code>MINUTE_OF_HOUR</code>	Минута часа	<code>mm: 02</code>

Окончание табл. 6.8

Константа перечислимого типа <code>ChronoField</code> или назначение	Перевод	Примеры
<code>SECOND_OF_MINUTE</code>	Секунда минуты	<code>ss: 00</code>
<code>NANO_OF_SECOND</code>	Наносекунда секунды	<code>nnnnnn: 000000</code>
Идентификатор часового пояса		<code>VV: America/New_York</code>
Наименование часового пояса		<code>z: EDT, zzzz: Eastern Daylight Time</code>
Смещение часового пояса		<code>x: -04, xx: -0400, xxx: -04:00, xxxx: то же самое, но Z обозначает нуль</code>
Локализованное смещение часового пояса		<code>O: GMT-4,0000: GMT-04:00</code>

Для синтаксического анализа значения даты и времени из символьной строки служит статический метод `parse()`, как показано ниже. В первом вызове этого метода используется стандартное средство форматирования `ISO_LOCAL_DATE`, а во втором вызове — специальное средство форматирования.

```
LocalDate churchsBirthday = LocalDate.parse("1903-06-14");
ZonedDateTime apollo11launch =
    ZonedDateTime.parse("1969-07-16 03:32:00-0400",
        DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ssxx"));
```

Порядок форматирования дат и времени демонстрируется в примере программы из листинга 6.4.

Листинг 6.4. Исходный код из файла `formatting/Formatting.java`

```
1 package formatting;
2
3 import java.time.*;
4 import java.time.format.*;
5 import java.util.*;
6
7 public class Formatting
8 {
9     public static void main(String[] args)
10    {
11        ZonedDateTime apollo11launch = ZonedDateTime.of(
12            1969, 7, 16, 9, 32, 0, 0, ZoneId.of("America/New_York"));
13
14        String formatted = DateTimeFormatter.ISO_OFFSET_DATE_TIME
15                    .format(apollo11launch);
16        // 1969-07-16T09:32:00-04:00
17        System.out.println(formatted);
18
19        DateTimeFormatter formatter = DateTimeFormatter
20                    .ofLocalizedDateTime(FormatStyle.LONG);
21        formatted = formatter.format(apollo11launch);
22        // July 16, 1969 9:32:00 AM EDT
23        System.out.println(formatted);
24        formatted = formatter.withLocale(Locale.FRENCH)
25                    .format(apollo11launch);
```

```

26     // 16 juillet 1969 09:32:00 EDT
27     System.out.println(formatted);
28
29     formatter = DateTimeFormatter
30         .ofPattern("E yyyy-MM-dd HH:mm");
31     formatted = formatter.format(apollo11launch);
32     System.out.println(formatted);
33
34     LocalDate churchsBirthday = LocalDate.parse("1903-06-14");
35     System.out.println("churchsBirthday: " + churchsBirthday);
36     apollo11launch = ZonedDateTime.parse(
37         "1969-07-16 03:32:00-0400",
38         DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ssxx"));
39     System.out.println("apollo11launch: " + apollo11launch);
40
41     for (DayOfWeek w : DayOfWeek.values())
42         System.out.print(w.getDisplayName(
43             TextStyle.SHORT, Locale.ENGLISH) + " ");
44     }
45 }
```

6.7. Взаимодействие с унаследованным кодом

Новый прикладной программный интерфейс Java API даты и времени должен обеспечивать нормальное взаимодействие с уже имеющимися классами. К их числу относятся широко распространенные классы `java.util.Date`, `java.util.GregorianCalendar` и `java.sql.Date/Time/Timestamp`.

Класс `Instant` очень похож на класс `java.util.Date`. В версии Java 8 этот класс был дополнен двумя методами. В частности, метод `toInstant()` служит для преобразования типа `Date` в тип `Instant`, а статический метод `from()` — для обратного преобразования этих типов даты и времени.

Аналогично класс `ZonedDateTime` очень похож на класс `java.util.GregorianCalendar` и дополнен соответствующими методами преобразования в версии Java 8. В частности, метод `toZonedDateTime()` служит для преобразования типа `GregorianCalendar` в тип `ZonedDateTime`, а метод `from()` — для обратного преобразования этих типов даты и времени.

Еще один ряд преобразований предусмотрен для классов даты и времени из пакета `java.sql`. Кроме того, средство форматирования типа `DateTimeFormatter` можно передать унаследованному коду, где применяется класс `java.text.Format`. Все эти методы преобразования сведены в табл. 6.9.

Таблица 6.9. Методы взаимного преобразования классов из пакета `java.time` и унаследованного кода

Классы	Метод преобразования в унаследованный код	Метод преобразования из унаследованного кода
<code>Instant ↔ java.util.GregorianCalendar</code>	<code>Date.from(instant)</code>	<code>date.toInstant()</code>
<code>ZonedDateTime ↔ java.util.GregorianCalendar</code>	<code>GregorianCalendar.from(zonedDateTime)</code>	<code>cal.toZonedDateTime()</code>

Окончание табл. 6.9

Классы	Метод преобразования в унаследованный код	Метод преобразования из унаследованного кода
Instant ↔ java.sql.Timestamp	TimeStamp.from(instant)	timestamp.toInstant()
LocalDateTime ↔ java.sql.Timestamp	Timestamp. valueOf(localDateTime)	timeStamp.toLocalDateTime()
LocalDate ↔ java.sql.Date	Date.valueOf(localDate)	date.toLocalDate()
LocalTime ↔ java.sql.Time	Time.valueOf(localTime)	time.toLocalTime()
DateTimeFormatter → java.text.DateFormat	formatter.toFormat()	Отсутствует
java.util.TimeZone → ZoneId	Timezone.getTimeZone(id)	timeZone.toZoneId()
java.nio.file. attribute.FileTime → Instant	FileTime.from(instant)	fileTime.toInstant()

Из этой главы вы узнали, как пользоваться библиотекой даты и времени, внедренной в версии Java 8, чтобы оперировать значениями даты и времени повсюду в мире. А в следующей главе речь пойдет об интернационализации прикладных программ на Java. В ней будет показано, как форматировать сообщения, выводимые программой, числа и денежные суммы в привычном для пользователей виде, где бы они ни находились.

Интернационализация

В этой главе...

- ▶ Региональные настройки
- ▶ Числовые форматы
- ▶ Форматирование денежных сумм в разных валютах
- ▶ Форматирование даты и времени
- ▶ Сортировка и нормализация
- ▶ Форматирование сообщений
- ▶ Ввод-вывод текста
- ▶ Комплекты ресурсов
- ▶ Пример интернационализации прикладной программы

Создавая приложение или аплет, разработчики надеются, что их программным продуктом заинтересуются многие пользователи. К тому же благодаря Интернету преодолеваются границы между разными странами. С другой стороны, если разработчики не принимают во внимание международных пользователей, то сами создают искусственные препятствия для широкого распространения своих программных продуктов по всему миру.

Java стал первым языком программирования, в котором изначально были предусмотрены средства интернационализации. Строки в Java формируются из символов в Юникоде (Unicode). Поддержка этого стандарта кодирования символов позволяет разрабатывать программы на Java, способные обрабатывать тексты на любом из языков, существующих в мире.

Многие программисты считают, что для интернационализации своих приложений им достаточно воспользоваться Юникодом и перевести на нужный язык все сообщения пользовательского интерфейса. Но этого явно недостаточно, потому что интернационализация программы означает нечто большее, чем поддержка кодировки символов в Юникоде. Дата, время, денежные суммы и даже числа

могут по-разному представляться на различных языках. Необходимо также найти простой способ настройки команд меню, надписей на кнопках, сообщений и комбинаций клавиш на выбранный язык и региональный стандарт.

В этой главе рассматриваются способы интернационализации прикладных программ на Java, а также особенности представления времени, даты, чисел, текста и элементов ГПИ с учетом региональных стандартов. Кроме того, в ней обсуждаются некоторые инструментальные средства, предназначенные для интернационализации программ. А в конце главы в качестве примера будет рассмотрена программа калькуляции пенсионных сбережений с пользовательским интерфейсом на английском, немецком и китайском языках.

7.1. Региональные настройки

Приложение, которое адаптировано для международного рынка, легко определить по возможности выбора языка, используемого для работы с ним. Но профессионально адаптированные приложения могут иметь разные региональные настройки даже для тех стран, в которых используется одинаковый язык. Эту ситуацию очень точно подметил некогда Оскар Уайльд: “Теперь у нас, действительно, все, как в Америке, естественно, кроме языка”.

В любом случае команды меню, надписи на кнопках и программные сообщения должны быть переведены на местный язык, возможно, с использованием другого алфавита. Существует еще много других, более тонких различий. Например, в Англии и Германии для представления десятичных чисел применяются разные форматы. Число, понятное англичанам в формате 123, 456.78, должно быть отображено в формате 123.456,78 для пользователей из Германии. Иными словами, точка и запятая в качестве разделителя дробной части и разделителя групп *по-разному* используются в этих странах!

Похожие различия можно заметить и в способах представления даты. В США привыкли отображать даты в формате месяц/день/год, в Германии используют более практичный порядок — день/месяц/год, а в Китае все наоборот — год/месяц/день. Таким образом, дата 3/22/61 должна быть представлена в формате 22.03.1961 для немецкого пользователя. Безусловно, если названия месяцев записаны полностью, различие в способах представления дат становится еще очевидным. Например, дата March 22, 1961, понятная для американского пользователя, должна быть представлена как 22. März 1961 для немецкого пользователя.

Имеется ряд классов, выполняющих форматирование и принимающих во внимание упомянутые выше региональные отличия. Для управления процессом форматирования служит класс `Locale`. *Региональные настройки* содержат следующие составляющие.

1. Язык, обозначаемый двумя или тремя строчными буквами, например, `en` (английский), `de` (немецкий) или `zh` (китайский). Наиболее употребительные коды языков перечислены в табл. 7.1.
2. Дополнительно письмо, обозначаемое четырьмя буквами, первая из которых является заглавной, например `Latn` (латынь), `Cyrl` (кириллица) или `Hant` (традиционные китайские иероглифы). Это удобно, поскольку в ряде

языков (например, в сербском) употребляется как латынь, так и кириллица, а некоторые китайские пользователи предпочитают традиционные иероглифы упрощенным.

3. Дополнительно страна или регион, обозначаемые двумя прописными буквами или тремя цифрами, например US (Соединенные Штаты) или CN (Швейцария). Наиболее употребительные коды стран перечислены в табл. 7.2.
4. Дополнительно *вариант*, обозначающий различные свойства языка, в том числе диалекты или правила произношения. Ныне варианты употребляются редко. Раньше употреблялся "новонорвежский" вариант норвежского языка, но теперь он выражается отдельным кодом языка nn. А прежние варианты японского императорского календаря и тайских цифр теперь выражаются как расширения, поясняемые ниже.
5. Дополнительно расширение. Расширения описывают локальные установки для календарей (например, японского календаря), чисел (тайских цифр вместо арабских) и т.д. Некоторые из этих расширений определены в стандарте Unicode. Расширения начинаются с обозначения u- и продолжаются двухбуквенным кодом, указывающим назначение расширения: ca — для календаря, nu — для чисел и т.д. Например, расширение u-nu-thai обозначает употребление тайских цифр. Другие расширения совершенно произвольны и начинаются с обозначения x-, например x-java.

Правила для региональных настроек сформулированы в памятной записке BCP 47 "Best Current Practices" (Передовые современные методики) Рабочей группы инженерной поддержки Интернета (Internet Engineering Task Force; <http://tools.ietf.org/html/bcp47>). Более доступное краткое изложение этих правил можно найти по адресу www.w3.org/International/articles/language-tags.

Таблица 7.1. Наиболее употребительные коды языков по стандарту ISO 639-1

Язык	Код	Язык	Код
Китайский	zh	Японский	ja
Датский	da	Корейский	ko
Голландский	nl	Норвежский	no
Английский	en	Португальский	pt
Французский	fr	Испанский	es
Финский	fi	Шведский	sv
Итальянский	it	Турецкий	tr

Таблица 7.2. Наиболее употребительные коды стран по стандарту ISO 3166-1

Страна	Код	Страна	Код
Австрия	AT	Япония	JP
Бельгия	BE	Корея	KR
Канада	CA	Нидерланды	NL
Китай	CN	Норвегия	NO
Дания	DK	Португалия	PT
Финляндия	FI	Испания	ES
Германия	DE	Швеция	SE

Окончание табл. 7.2

Страна	Код	Страна	Код
Великобритания	GB	Швейцария	CH
Греция	GR	Тайвань	TW
Ирландия	IE	Турция	TR
Италия	IT	Соединенные Штаты	US

Коды языков и стран кажутся на первый взгляд выбранными несколько произвольно, поскольку некоторые из них происходят от местных языков. Так, *Deutsch* по-немецки означает "немецкий", а *zhongwen* по-китайски (в латинской транскрипции) — "китайский", отсюда и коды этих языков *de* и *zh* соответственно. Швейцария обозначается кодом *CH*, происходящим об латинского термина *Confoederatio Helvetica*, означающего "Швейцарская конфедерация".

Региональные настройки описываются дескрипторами в виде символьных строк, где элементы региональных настроек указываются через дефис, например, *en-US* для США или *de-DE* для Германии. А в Швейцарии приняты четыре официальных языка (немецкий, французский, итальянский и ретороманский), поэтому для немецкоязычного пользователя из Швейцарии потребуются региональные настройки *de-CH*, соблюдающие правила немецкого языка, но выражающие денежные суммы в швейцарских франках, а не в евро. Если же указать только язык (*de*), то такие региональные настройки нельзя использовать для выражения характерных для конкретной страны особенностей вроде представления денежных сумм в местной валюте.

Создать объект типа *Locale* из символьной строки дескриптора можно следующим образом:

```
Locale usEnglish = Locale.forLanguageTag("en-US");
```

Метод *toLanguageTag()* возвращает дескриптор языка из заданных региональных настроек. Так, в результате вызова *Locale.US.toLanguageTag()* возвращается символьная строка "en-US".

Ради удобства для различных стран заранее определены следующие объекты региональных настроек:

```
Locale.CANADA
Locale.CANADA_FRENCH
Locale.CHINA
Locale.FRANCE
Locale.GERMANY
Locale.ITALY
Locale.JAPAN
Locale.KOREA
Locale.PRC
Locale.TAIWAN
Locale.UK
Locale.US
```

Ряд заранее определенных региональных настроек обозначают только язык, но не страну или регион, как показано ниже. И наконец, статический метод *getAvailableLocales()* возвращает массив всех региональных настроек, известных виртуальной машине.

```
Locale.CHINESE
Locale.ENGLISH
```

```
Locale.FRENCH
Locale.GERMAN
Locale.ITALIAN
Locale.JAPANESE
Locale.KOREAN
Locale.SIMPLIFIED_CHINESE
Locale.TRADITIONAL_CHINESE
```

Помимо вызова конструктора или выбора предопределенных объектов, имеются еще два способа для получения объектов с региональными настройками. В частности, статический метод `getDefault()` из класса `Locale` позволяет определить региональные настройки, которые используются в операционной системе по умолчанию. Изменить эти настройки по умолчанию можно, вызвав метод `setDefault()`, но не следует забывать, что действие этого метода распространяется только на прикладную программу на Java, а не на всю операционную систему в целом.

Все зависимые от региональных стандартов вспомогательные классы могут возвращать массив поддерживаемых региональных настроек. Например, приведенный ниже метод возвращает все региональные настройки, поддерживаемые в классе `DateFormat`.

```
Locale[] supportedLocales = DateFormat.getAvailableLocales();
```



Совет! Для проверки интернационализации своей программы попробуйте сменить региональные настройки по умолчанию. С этой целью укажите язык и страну при запуске программы на выполнение из командной строки. Например, в приведенной ниже команде запуска программы задается немецкий язык для Швейцарии.

```
java -Duser.language=de -Duser.region=CH МояПрограмма
```

Что же можно сделать с полученными региональными настройками? Оказывается, не так уж и много. Единственную пользу можно извлечь из тех методов в классе `Locale`, которые определяют коды языка и страны. Наиболее важным из них является метод `getDisplayName()`, который возвращает символьную строку с описанием региональной настройки. Она содержит не какие-то загадочные двухбуквенные коды, а вполне удобочитаемое описание, как в приведенном ниже примере.

```
German (Switzerland)
```

Но дело в том, что эта строка отображается на используемом по умолчанию языке, что далеко не всегда удобно. Так, если пользователь выбрал немецкий язык интерфейса, символьную строку описания региональной настройки следует отобразить именно на немецком языке, передав в качестве параметра соответствующую региональную настройку:

```
Locale loc = new Locale("de", "CH");
System.out.println(loc.getDisplayName(Locale.GERMAN));
```

В результате выполнения этого фрагмента кода описание региональной настройки будет выведено на указанном в ней языке:

```
Deutsch (Schweiz)
```

Из данного примера становится ясно, зачем нужны объекты типа `Locale`. Передавая их методам, способным реагировать на региональные настройки, можно

отображать текст на языке, понятном пользователю. Примеры применения таких объектов рассматриваются далее в этой главе.

java.util.Locale 1.1

- **Locale(String language)**
Создают объект типа **Locale** региональных настроек с учетом указанного языка, страны и варианта языка. Вместо вариантов языка в новом коде рекомендуется использовать языковые дескрипторы по стандарту IETF BCP 47.
- **static Locale forLanguageTag(String languageTag) 7**
Создает объект типа **Locale** региональных настроек по заданному языковому дескриптору.
- **static Locale getDefault()**
Возвращает региональные настройки, используемые по умолчанию.
- **static void setDefault(Locale loc)**
Задает региональные настройки, используемые по умолчанию.
- **String getDisplayName()**
Возвращает символьную строку наименования с описанием текущей региональной настройки. Эта строка составляется на том языке, который указан в текущих региональных настройках.
- **String getDisplayName(Locale loc)**
Возвращает символьную строку наименования с описанием заданных региональных настроек. Эта строка составляется на том языке, который указан в заданных региональных настройках.
- **String getLanguage()**
Возвращает код языка [две строчные буквы] по стандарту ISO 639-1.
- **String getDisplayLanguage()**
Возвращает название языка в формате текущих региональных настроек.
- **String getDisplayLanguage(Locale loc)**
Возвращает название языка в формате заданных региональных настроек.
- **String getCountry()**
Возвращает код страны [две прописные буквы] по стандарту ISO 3166-1.
- **String getDisplayCountry()**
Возвращает название страны в формате текущих региональных настроек.
- **String getDisplayCountry(Locale loc)**
Возвращает название страны в формате заданных региональных настроек.
- **String toLanguageTag() 7**
Возвращает языковой дескриптор по стандарту IETF BCP 47 для текущих региональных настроек, например "de-CH" (немецкий язык, Швейцария).
- **String toString()**
Возвращает описание региональных настроек в виде кодов языка и страны, разделенных знаком подчеркивания (например, "de-CH", т.е. немецкий язык, Швейцария). Этим методом рекомендуется пользоваться только для целей отладки.

7.2. Числовые форматы

Как упоминалось выше, в разных странах и регионах применяются различные способы представления чисел и денежных сумм. В пакете `java.text` содержатся классы, позволяющие форматировать числа и выполнять синтаксический анализ их строкового представления. Для форматирования числа в соответствии с конкретными региональными настройками необходимо выполнить следующие действия.

1. Получить объект региональных настроек, как пояснялось в предыдущем разделе.
2. Вызвать фабричный метод для получения форматирующего объекта.
3. Применить форматирующий объект для форматирования числа или синтаксического анализа его строкового представления.

В качестве фабричных служат статические методы `getNumberInstance()`, `getCurrencyInstance()` и `getPercentInstance()` из класса `NumberFormat`. Они получают в качестве параметра объект типа `Locale` и возвращают объекты, предназначенные для форматирования чисел, денежных сумм и числовых величин в процентах. Например, для выражения денежной суммы в формате, принятом в Германии, служит приведенный ниже фрагмент кода.

```
Locale loc = new Locale("de", "DE");
NumberFormat currFmt = NumberFormat.getCurrencyInstance(loc);
double amt = 123456.78;
String result = currFmt.format(amt);
```

В результате выполнения этого фрагмента кода получается следующая символьная строка:

123.456,78 €

Для обозначения европейской валюты в данном случае используется знак **€**, который располагается в конце строки. Обратите также внимание на местоположение знаков, обозначающих дробную часть числа и разделяющих десятичные разряды в его целой части. Порядок их следования обратный принятому в англоязычных странах.

Рассмотрим обратную задачу преобразования в число символьной строки, составленной в соответствии с конкретными региональными настройками. Для этого предусмотрен метод `parse()`, выполняющий синтаксический анализ символьной строки, автоматически используя заданные по умолчанию региональные настройки. В приведенном ниже примере кода демонстрируется преобразование в число символьной строки, введенной пользователем в текстовом поле. Метод `parse()` способен преобразовывать символьные строки в числа, где в качестве разделителей используются точки и запятые в соответствии с принятыми региональными стандартами.

```
TextField inputField;
.
.
.
NumberFormat fmt = NumberFormat.getNumberInstance();
// получить средство форматирования чисел для используемых
// по умолчанию региональных настроек
```

```
Number input = fmt.parse(inputField.getText().trim());  
double x = input.doubleValue();
```

Метод `parse()` возвращает результат абстрактного типа `Number`. На самом деле возвращаемый объект является экземпляром класса `Long` или `Double` в зависимости от того, представляет ли исходная символьная строка целое число или же число с плавающей точкой. Если это не так важно, то для получения упакованного числового значения достаточно вызвать метод `doubleValue()` из класса `Number`.



Внимание! Для объектов типа `Number` не поддерживается автоматическая распаковка, поэтому их нельзя присвоить непосредственно переменным примитивных типов. Вместо этого нужно явным образом вызвать метод `doubleValue()` или `intValue()`.

Если число представлено в неверном формате, генерируется исключение типа `ParseException`. Например, не допускается наличие пробелов в начале символьной строки, преобразуемой в число. (Для их удаления следует вызвать метод `trim()`.) Но любые символы, следующие в строке после числа, просто игнорируются, и поэтому исключение в этом случае не возникает.

Следует, однако, иметь в виду, что классы, возвращаемые фабричными методами типа `getXXXInstance()`, являются экземплярами не абстрактного класса `NumberFormat`, а одного из его подклассов. Фабричным методам известно только, как найти объект, относящийся к конкретным региональным настройкам.

Для получения списка поддерживаемых региональных настроек можно вызвать статический метод `getAvailableLocales()`. Этот метод возвращает массив региональных настроек, для которых могут быть получены форматирующие объекты.

В примере программы, рассматриваемой в этом разделе, предоставляется возможность поэкспериментировать с разными способами форматирования чисел, как показано на рис. 7.1. В верхней части рабочего окна этой программы находится список всех доступных региональных настроек со средствами форматирования чисел. Ниже этого списка расположена группа кнопок-переключателей, где можно выбрать способ форматирования чисел, денежных сумм или числовых величин в процентах. После выбора новой региональной настройки или способа форматирования чисел число в текстовом поле автоматически переформатируется.

Просмотрев лишь несколько вариантов форматирования чисел и денежных сумм в зависимости от выбранных региональных настроек, пользователь, несомненно, составит ясное представление о разнообразии существующих форматов чисел. В текстовом поле можно ввести любое число и щелкнуть на кнопке `Parse`, в результате чего будет вызван метод `parse()`, выполняющий синтаксический анализ введенной символьной строки. При удачном исходе синтаксического анализа результат передается методу `format()`, а затем отображается на экране. В противном случае в текстовом поле появляется сообщение "Parse error" (Ошибка синтаксического анализа).

Как показано в листинге 7.1, исходный код рассматриваемой здесь программы имеет довольно простую структуру. Сначала в конструкторе вызывается метод `NumberFormat.getAvailableLocales()`. Затем для каждого вида поддерживаемых региональных настроек вызывается метод `getDisplayNames()`, а возвращаемые этим методом результаты вводятся в список. (Символьные строки не сортируются; подробнее об этом речь пойдет в разделе 7.5.) Всякий раз, когда

пользователь выбирает другие региональные настройки или способ форматирования чисел, создается новый форматирующий объект и обновляется содержимое текстового поля. Если пользователь щелкнет на кнопке Parse, то вызывается метод parse(), преобразующий в число символьную строку в соответствии с выбранными региональными настройками.

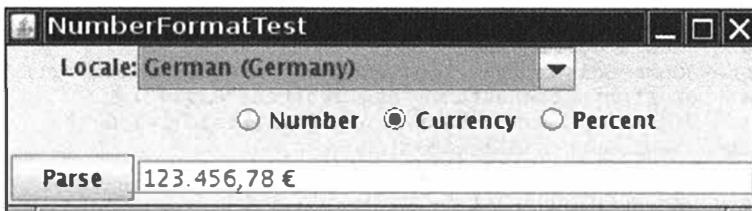


Рис. 7.1. Рабочее окно программы NumberFormatTest

На заметку! Для чтения локализованных целых чисел, а также чисел с плавающей точкой можно воспользоваться классом `Scanner`, вызвав метод `useLocale()` из этого класса для установки региональных настроек.

Листинг 7.1. Исходный код из файла numberFormat/NumberFormatTest.java

```

1 package numberFormat;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.text.*;
6 import java.util.*;
7
8 import javax.swing.*;
9
10 /**
11 * В этой программе демонстрируется форматирование чисел
12 * при разных региональных настройках
13 * @version 1.14 2016-05-06
14 * @author Cay Horstmann
15 */
16 public class NumberFormatTest
17 {
18     public static void main(String[] args)
19     {
20         EventQueue.invokeLater(() ->
21         {
22             JFrame frame = new NumberFormatFrame();
23             frame.setTitle("NumberFormatTest");
24             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25             frame.setVisible(true);
26         });
27     }
28 }
29
30 /**
31 * Этот фрейм содержит кнопки-переключатели для выбора способа

```

```
32 * форматирования чисел, комбинированный список для выбора
33 * региональных настроек, текстовое поле для отображения
34 * отформатированного числа, а также кнопку для активизации
35 * синтаксического анализа содержимого текстового поля
36 */
37 class NumberFormatFrame extends JFrame
38 {
39     private Locale[] locales;
40     private double currentNumber;
41     private JComboBox<String> localeCombo = new JComboBox<>();
42     private JButton parseButton = new JButton("Parse");
43     private JTextField numberText = new JTextField(30);
44     private JRadioButton numberRadioButton =
45         new JRadioButton("Number");
46     private JRadioButton currencyRadioButton =
47         new JRadioButton("Currency");
48     private JRadioButton percentRadioButton =
49         new JRadioButton("Percent");
50     private ButtonGroup rbGroup = new ButtonGroup();
51     private NumberFormat currentNumberFormat;
52
53     public NumberFormatFrame()
54     {
55         setLayout(new GridBagLayout());
56
57         ActionListener listener = event -> updateDisplay();
58
59         JPanel p = new JPanel();
60         addRadioButton(p, numberRadioButton, rbGroup, listener);
61         addRadioButton(p, currencyRadioButton, rbGroup, listener);
62         addRadioButton(p, percentRadioButton, rbGroup, listener);
63
64         add(new JLabel("Locale:"), 
65             new GBC(0, 0).setAnchor(GBC.EAST));
66         add(p, new GBC(1, 1));
67         add(parseButton, new GBC(0, 2).setInsets(2));
68         add(localeCombo, new GBC(1, 0).setAnchor(GBC.WEST));
69         add(numberText, new GBC(1, 2).setFill(GBC.HORIZONTAL));
70         locales =
71             (Locale[]) NumberFormat.getAvailableLocales().clone();
72         Arrays.sort(locales,
73             Comparator.comparing(Locale::getDisplayName));
74         for (Locale loc : locales)
75             localeCombo.addItem(loc.getDisplayName());
76         localeCombo.setSelectedItem(
77             Locale.getDefault().getDisplayName());
78         currentNumber = 123456.78;
79         updateDisplay();
80
81         localeCombo.addActionListener(listener);
82
83         parseButton.addActionListener(event ->
84         {
85             String s = numberText.getText().trim();
86             try
87             {
88                 Number n = currentNumberFormat.parse(s);
89                 if (n != null)
```

```
90         {
91             currentNumber = n.doubleValue();
92             updateDisplay();
93         }
94     else
95     {
96         numberText.setText("Parse error: " + s);
97     }
98 }
99 catch (ParseException e)
100 {
101     numberText.setText("Parse error: " + s);
102 }
103 });
104 pack();
105 }
106
107 /**
108 * Вводит кнопки-переключатели в контейнер
109 * @param p Контейнер для размещения кнопок-переключателей
110 * @param b Кнопка-переключатель
111 * @param g Группа кнопок-переключателей
112 * @param listener Приемник событий от кнопок-переключателей
113 */
114 public void addRadioButton(Container p, JRadioButton b,
115                             ButtonGroup g, ActionListener listener)
116 {
117     b.setSelected(g.getButtonCount() == 0);
118     b.addActionListener(listener);
119     g.add(b);
120     p.add(b);
121 }
122
123 /**
124 * Обновляет отображаемое число и форматирует его
125 * в соответствии с пользовательскими установками
126 */
127 public void updateDisplay()
128 {
129     Locale currentLocale =
130         locales[localeCombo.getSelectedIndex()];
131     currentNumberFormat = null;
132     if (numberRadioButton.isSelected())
133         currentNumberFormat =
134             NumberFormat.getNumberInstance(currentLocale);
135     else if (currencyRadioButton.isSelected())
136         currentNumberFormat =
137             NumberFormat.getCurrencyInstance(currentLocale);
138     else if (percentRadioButton.isSelected())
139         currentNumberFormat =
140             NumberFormat.getPercentInstance(currentLocale);
141     String formatted = currentNumberFormat.format(currentNumber);
142     numberText.setText(formatted);
143 }
144 }
```

java.text.NumberFormat 1.1

- **static Locale[] getAvailableLocales()**

Возвращает массив объектов типа **Locale**, для которых доступны форматирующие объекты типа **NumberFormat**.

- **static NumberFormat getInstance()**

- **static NumberFormat getInstance(Locale l)**

- **static NumberFormat getCurrencyInstance()**

- **static NumberFormat getCurrencyInstance(Locale l)**

- **static NumberFormat getPercentInstance()**

- **static NumberFormat getPercentInstance(Locale l)**

Возвращают объект, форматирующий числа, денежные суммы или числовые величины в процентах в соответствии с текущими или заданными региональными настройками.

- **String format(double x)**

- **String format(long x)**

Возвращают символьную строку, получаемую в результате форматирования заданного числа с плавающей точкой или целого числа.

- **Number parse(String s)**

Возвращает число, получаемое в результате синтаксического анализа символьной строки. Это число может иметь тип **Long** или **Double**, а символьная строка не должна начинаться с пробелов. Любые символы, следующие в строке после анализируемого числа, игнорируются. При неудачном исходе синтаксического анализа символьной строки генерируется исключение типа **ParseException**.

- **void setParseIntegerOnly(boolean b)**

- **boolean isParseIntegerOnly()**

Устанавливают или получают признак, указывающий на то, что данный форматирующий объект предназначен для синтаксического анализа только целочисленных значений.

- **void setGroupingUsed(boolean b)**

- **boolean isGroupingUsed()**

Устанавливают или получают признак, указывающий на то, что данный форматирующий объект предназначен для распознавания и разделения групп десятичных разрядов (например, 100,000) в анализируемых числах.

- **void setMinimumIntegerDigits(int n)**

- **int getMinimumIntegerDigits()**

- **void setMaximumIntegerDigits(int n)**

- **int getMaximumIntegerDigits()**

- **void setMinimumFractionDigits(int n)**

- **int getMinimumFractionDigits()**

- **void setMaximumFractionDigits(int n)**

- **int getMaximumFractionDigits()**

Устанавливают или получают максимальное или минимальное количество цифр в целой или дробной части числа.

7.3. Форматирование денежных сумм в разных валютах

Для форматирования денежных сумм служит метод `NumberFormat.getCurrencyInstance()`, но он не очень удобен, поскольку возвращает форматирующий объект только для одной валюты. Допустим, что для американского заказчика выписывается счет-фактура, где одни суммы представлены в долларах США, а другие в евро. Для решения этой задачи нельзя воспользоваться просто двумя форматирующими объектами, как показано ниже. Счет, в котором фигурируют такие суммы, как \$100,000 и 100.000€, будет выглядеть не совсем обычно. Ведь при представлении сумм в евро для разделения групп разрядов используется точка, а сумм в долларах США — запятая.

```
NumberFormat dollarFormatter =
    NumberFormat.getCurrencyInstance(Locale.US);
NumberFormat euroFormatter =
    NumberFormat.getCurrencyInstance(Locale.GERMANY);
```

Для управления форматированием денежных сумм в разных валютах лучше воспользоваться классом `Currency`. Сначала получается объект типа `Currency`, для чего статическому методу `Currency.getInstance()` передается идентификатор валюты. Затем вызывается метод `setCurrency()` для каждого форматирующего объекта. В приведенном ниже фрагменте кода показано, как подстроить объект, форматирующий денежные суммы в евро, под американского заказчика.

```
NumberFormat euroFormatter =
    NumberFormat.getCurrencyInstance(Locale.US);
euroFormatter.setCurrency(Currency.getInstance("EUR"));
```

Идентификаторы валют определяются по стандарту ISO 4217 (<http://www.currency-iso.org/en/home/tables/table-a1.html>). Некоторые из них приведены в табл. 7.3.

Таблица 7.3. Идентификаторы валют

Валюта	Идентификатор
Доллар США	USD
Евро	EUR
Английский фунт	GBP
Японская иена	JPY
Китайский юань	CNY
Индийская рупия	INR
Российский рубль	RUB

java.util.Currency 1.4

- `static Currency getInstance(String currencyCode)`
- `static Currency getInstance(Locale locale)`

Возвращают экземпляр класса `Currency`, соответствующий заданному коду валюты по стандарту ISO 4217 или стране, указанной в текущих региональных настройках.

java.util.Currency 1.4 (окончание)

- **String toString()**
- **String getCurrencyCode()**
Получают код текущей валюты по стандарту ISO 4217.
- **String getSymbol()**
- **String getSymbol(Locale locale)**
Получают форматирующий знак текущей валюты в соответствии с текущими или заданными региональными настройками. Например, доллар США (**USD**) может обозначаться как \$ или US\$ в зависимости от используемых региональных настроек.
- **int getDefaultFractionDigits()**
Получает принятное по умолчанию количество цифр в дробной части денежной суммы, указанной в текущей валюте.
- **static Set<Currency> getAvailableCurrencies() 7**
Получает все имеющиеся валюты.

7.4. Форматирование даты и времени

При форматировании даты и времени в соответствии с региональными настройками необходимо иметь в виду четыре особенности.

- Названия месяцев и дней недели должны быть представлены на местном языке.
- Порядок указания года, месяца и числа отличается в разных странах и регионах.
- Для отображения дат может использоваться календарь, отличный от григорианского.
- Следует учитывать часовые пояса.

Для форматирования даты и времени применяется класс `DateTimeFormatter`. Затем выбирается один из четырех стилей форматирования, перечисленных в табл. 7.4. Далее получается средство форматирования:

```
FormatStyle style = ...; // Один из стилей форматирования
                        // FormatStyle.SHORT, FormatStyle.MEDIUM, ...
DateTimeFormatter dateFormatter =
    DateTimeFormatter.ofLocalizedDate(style);
DateTimeFormatter timeFormatter =
    DateTimeFormatter.ofLocalizedTime(style);
DateTimeFormatter dateTimeFormatter =
    DateTimeFormatter.ofLocalDateTime(style);
    // или DateTimeFormatter.ofLocalDateTime(style1, style2)
```

В этих средствах форматирования используются текущие региональные настройки форматов даты и времени. Чтобы выбрать другие региональные настройки, достаточно вызвать метод `withLocale()` следующим образом:

```
DateTimeFormatter dateFormatter =
    DateTimeFormatter.ofLocalizedDate(style).withLocale(locale);
```

Таблица 7.4. Стили форматирования даты и времени с учетом региональных настроек

Стиль	Дата	Время
SHORT	7/16/69	9:32 AM
MEDIUM	Jul 16, 1969	9:32:00 AM
LONG	July 16, 1969	9:32:00 AM EDT с учетом региональных настроек en-US, 9:32:00 MSZ с учетом региональных настроек de-DE (только для класса ZonedDateTime)
FULL	Wednesday, July 16, 1969	9:32:00 AM EDT с учетом региональных настроек en-US, 9:32 Uhr MSZ с учетом региональных настроек de-DE (только для класса ZonedDateTime)

Теперь можно отформатировать местную дату (объект типа LocalDate), местное время и дату (объект типа LocalDateTime), местное время (объект типа LocalTime) или поясное время и дату (объект типа ZonedDateTime), как показано ниже.

```
ZonedDateTime appointment = ...;
String formatted = formatter.format(appointment);
```



На заметку! В данном случае применяется класс `DateTimeFormatter` из пакета `java.time`. Имеется также устаревший класс `java.text.SimpleDateFormat`, внедренный еще в версии Java 1.1 для манипулирования объектами типа `Date` и `Calendar`.

Для синтаксического анализа символьной строки, содержащей дату и время, служит один из статических методов `parse()` в классе `LocalDate`, `LocalDateTime`, `LocalTime` или `ZonedDateTime`:

```
LocalTime time = LocalTime.parse("9:32 AM", formatter);
```

Но методы `parse()` из упомянутых выше классов непригодны для синтаксического анализа данных, вводимых пользователем, — по крайней мере, для предварительной их обработки. Например, средство форматирования даты и времени в кратком стиле для Соединенных Штатов способно проанализировать символьную строку "9:32 AM", но не строку "9:32AM" или "9:32 ам".



Внимание! Средства форматирования дат подвергают синтаксическому анализу несуществующие даты вроде 31 ноября, корректируя их по последней дате в данном месяце.

Иногда в календарном приложении требуется отображать, например, только наименования дней недели и месяцы. С этой целью можно вызвать метод `getDisplayName()` из перечислений `DayOfWeek` и `Month`, как показано ниже.

```
for (Month m : Month.values())
    System.out.println(m.getDisplayName(textStyle, locale) + " ");
```

Стили форматирования текста перечислены в табл. 7.5. Стили типа STANDALONE служат для отображения за пределами форматируемой даты. Например, январь по-фински обозначается как "tammikuuta" в самой дате, но как "tammikuu" за ее пределами или отдельно.

Таблица 7.5. Стили форматирования текста, представленные константами из перечисления `java.time.format.TextStyle`

Стиль	Пример
<code>FULL / FULL_STANDALONE</code>	<code>January</code>
<code>SHORT / SHORT_STANDALONE</code>	<code>Jan</code>
<code>NARROW / NARROW_STANDALONE</code>	<code>J</code>

 **На заметку!** Первым днем недели может быть суббота, воскресенье или понедельник в зависимости от конкретных региональных настроек. Выяснить первый день недели с учетом региональных настроек можно следующим образом:

```
DayOfWeek first = WeekFields.of(locale).getFirstDayOfWeek();
```

В примере программы, исходный код которой приведен в листинге 7.2, демонстрируется применение класса `DateFormat` на практике. Эта программа позволяет выбирать различные региональные настройки и наблюдать за тем, как в разных странах мира форматируются дата и время. На рис. 7.2 показано рабочее окно данной программы после установки китайских шрифтов на компьютере. Как видите, даты выводятся на экран в правильном формате для китайских региональных настроек.

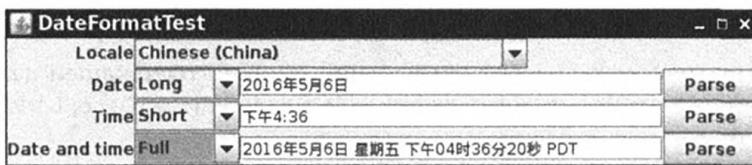


Рис. 7.2. Отображение даты на китайском языке
в рабочем окне программы `DateFormatTest`

Листинг 7.2. Исходный код из файла `dateFormat/DateFormatTest.java`

```

1 package dateFormat;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.time.*;
6 import java.time.format.*;
7 import java.util.*;
8
9 import javax.swing.*;
10
11 /**
12 * В этой программе демонстрируется форматирование дат
13 * при выборе разных региональных настроек
14 * @version 1.00 2016-05-06
15 * @author Cay Horstmann
16 */
17 public class DateTimeFormatterTest
18 {
19     public static void main(String[] args)
20     {

```

```
21     EventQueue.invokeLater(() ->
22     {
23         JFrame frame = new DateTimeFormatterFrame();
24         frame.setTitle("DateFormatTest");
25         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26         frame.setVisible(true);
27     });
28 }
29 }
30 */
31 /**
32 * Этот фрейм содержит комбинированные списки для выбора
33 * региональных настроек и форматов даты и времени, текстовые
34 * поля для отображения отформатированных даты и времени, а
35 * также кнопки синтаксического анализа содержимого текстовых
36 * полей и флагок для установки режима нестрогой интерпретации
37 * вводимых дат и времени
38 */
39 class DateTimeFormatterFrame extends JFrame
40 {
41     private Locale[] locales;
42     private LocalDate currentDate;
43     private LocalTime currentTime;
44     private ZonedDateTime currentDateTime;
45     private DateTimeFormatter currentDateFormat;
46     private DateTimeFormatter currentTimeFormat;
47     private DateTimeFormatter currentDateTimeFormat;
48     private JComboBox<String> localeCombo = new JComboBox<>();
49     private JButton dateParseButton = new JButton("Parse");
50     private JButton timeParseButton = new JButton("Parse");
51     private JButton dateTimeParseButton = new JButton("Parse");
52     private JTextField dateText = new JTextField(30);
53     private JTextField timeText = new JTextField(30);
54     private JTextField dateTimeText = new JTextField(30);
55     private EnumCombo<FormatStyle> dateStyleCombo =
56         new EnumCombo<>(FormatStyle.class, "Short",
57                           "Medium", "Long", "Full");
58     private EnumCombo<FormatStyle> timeStyleCombo =
59         new EnumCombo<>(FormatStyle.class, "Short", "Medium");
60     private EnumCombo<FormatStyle> dateTimeStyleCombo =
61         new EnumCombo<>(FormatStyle.class, "Short",
62                           "Medium", "Long", "Full");
62
63
64     public DateTimeFormatterFrame()
65     {
66         setLayout(new GridBagLayout());
67         add(new JLabel("Locale"), new GBC(0, 0).setAnchor(GBC.EAST));
68         add(localeCombo, new GBC(1, 0, 2, 1).setAnchor(GBC.WEST));
69
70         add(new JLabel("Date"), new GBC(0, 1).setAnchor(GBC.EAST));
71         add(dateStyleCombo, new GBC(1, 1).setAnchor(GBC.WEST));
72         add(dateText, new GBC(2, 1, 2, 1).setFill(GBC.HORIZONTAL));
73         add(dateParseButton, new GBC(4, 1).setAnchor(GBC.WEST));
74
75         add(new JLabel("Time"), new GBC(0, 2).setAnchor(GBC.EAST));
76         add(timeStyleCombo, new GBC(1, 2).setAnchor(GBC.WEST));
77         add(timeText, new GBC(2, 2, 2, 1).setFill(GBC.HORIZONTAL));
78         add(timeParseButton, new GBC(4, 2).setAnchor(GBC.WEST));
```

```
79      add(new JLabel("Date and time"),
80            new GBC(0, 3).setAnchor(GBC.EAST));
81      add(dateTextStyleCombo, new GBC(1, 3).setAnchor(GBC.WEST));
82      add(dateTimeText, new GBC(2, 3, 2, 1)
83                      .setFill(GBC.HORIZONTAL));
84      add(dateTimeParseButton, new GBC(4, 3).setAnchor(GBC.WEST));
85
86
87      locales = (Locale[]) Locale.getAvailableLocales().clone();
88      Arrays.sort(locales, Comparator
89                      .comparing(Locale::getDisplayName));
90      for (Locale loc : locales)
91          localeCombo.addItem(loc.getDisplayName());
92      localeCombo.setSelectedItem(Locale.getDefault()
93                      .getDisplayName());
94      currentDate = LocalDate.now();
95      currentTime = LocalTime.now();
96      currentDateTime = ZonedDateTime.now();
97      updateDisplay();
98
99      ActionListener listener = event -> updateDisplay();
100
101     localeCombo.addActionListener(listener);
102     dateStyleCombo.addActionListener(listener);
103     timeStyleCombo.addActionListener(listener);
104     dateTimeStyleCombo.addActionListener(listener);
105
106    dateParseButton.addActionListener(event ->
107    {
108        String d = dateText.getText().trim();
109        try
110        {
111            currentDate = LocalDate.parse(d, currentDateFormat);
112            updateDisplay();
113        }
114        catch (Exception e)
115        {
116            dateText.setText(e.getMessage());
117        }
118    });
119
120    timeParseButton.addActionListener(event ->
121    {
122        String t = timeText.getText().trim();
123        try
124        {
125            currentTime = LocalTime.parse(t, currentTimeFormat);
126            updateDisplay();
127        }
128        catch (Exception e)
129        {
130            timeText.setText(e.getMessage());
131        }
132    });
133
134    dateTimeParseButton.addActionListener(event ->
135    {
136        String t = dateTimeText.getText().trim();
```

```
137     try
138     {
139         currentDateTime = ZonedDateTime.parse(
140             t, currentDateTimeFormat);
141         updateDisplay();
142     }
143     catch (Exception e)
144     {
145         dateTimeText.setText(e.getMessage());
146     }
147 });
148
149     pack();
150 }
151
152 /**
153 * Обновляет отображаемые дату и время и форматирует
154 * их в соответствии с пользовательскими установками
155 */
156 public void updateDisplay()
157 {
158     Locale currentLocale = locales[localeCombo
159             .getSelectedIndex()];
160     FormatStyle dateStyle = dateStyleCombo.getValue();
161     currentDateFormat = DateTimeFormatter.ofLocalizedDate(
162         dateStyle).withLocale(currentLocale);
163     dateText.setText(currentDateFormat.format(currentDate));
164     FormatStyle timeStyle = timeStyleCombo.getValue();
165     currentTimeFormat = DateTimeFormatter.ofLocalizedTime(
166         timeStyle).withLocale(currentLocale);
167     timeText.setText(currentTimeFormat.format(currentTime));
168     FormatStyle dateTimeStyle = dateTimeStyleCombo.getValue();
169     currentDateTimeFormat = DateTimeFormatter
170             .ofLocalizedDateTime(dateTimeStyle)
171             .withLocale(currentLocale);
172     dateTimeText.setText(currentDateTimeFormat
173             .format(currentDateTime));
174 }
175 }
```

Для проверки правильности синтаксического анализа и преобразования символьной строки в дату достаточно ввести дату, время или и то и другое, а затем щелкнуть на кнопке Parse ((Произвести синтаксический анализ). В рассматриваемом здесь примере программы используется вспомогательный класс EnumCombo, исходный код которого приведен в листинге 7.3. Он служит для заполнения комбинированного списка значениями типа Short, Medium и Long, а также для автоматического преобразования выбранного пользователем варианта в значение FormatStyle.SHORT, FormatStyle.MEDIUM или FormatStyle.LONG. Чтобы не писать повторяющийся код, в данном случае применяется рефлексия. Выбранный пользователем вариант преобразуется в верхний регистр, пробелы заменяются символами подчеркивания, после чего определяется значение в статическом поле с полученным в итоге именем. (Более подробно рефлексия рассматривается в главе 5 первого тома настоящего издания.)

Листинг 7.3. Исходный код из файла dateFormat/EnumCombo.java

```
1 package dateFormat;
2
3 import java.util.*;
4 import javax.swing.*;
5
6 /**
7  * Комбинированный список для выбора среди значений
8  * статических полей, имена которых задаются в
9  * конструкторе вспомогательного класса
10 * @version 1.15 2016-05-06
11 * @author Cay Horstmann
12 */
13 public class EnumCombo<T> extends JComboBox<String>
14 {
15     private Map<String, T> table = new TreeMap<>();
16
17 /**
18  * Конструирует объект вспомогательного класса EnumCombo,
19  * производящий значения типа T
20  * @param cl Класс
21  * @param labels Массив символьных строк, описывающих имена
22  * статических полей из класса cl, относящихся к типу T
23  */
24 public EnumCombo(Class<?> cl, String... labels)
25 {
26     for (String label : labels)
27     {
28         String name = label.toUpperCase().replace(' ', '_');
29         try
30         {
31             java.lang.reflect.Field f = cl.getField(name);
32             @SuppressWarnings("unchecked") T value = (T) f.get(cl);
33             table.put(label, value);
34         }
35         catch (Exception e)
36         {
37             label = "(" + label + ")";
38             table.put(label, null);
39         }
40         addItem(label);
41     }
42     setSelectedItem(labels[0]);
43 }
44
45 /**
46  * Возвращает значение поля, выбранного пользователем
47  * @return Значение статического поля
48  */
49 public T getValue()
50 {
51     return table.get(getSelectedItem());
52 }
53 }
```

```
java.time.format.DateTimeFormatter 8
```

- **static DateTimeFormatter ofLocalizedDate(FormatStyle dateStyle)**
- **static DateTimeFormatter ofLocalizedTime(FormatStyle dateStyle)**
- **static DateTimeFormatter ofLocalDateTime(FormatStyle dateStyle, FormatStyle timeStyle)**

Возвращают экземпляры типа **DateTimeFormatter** для форматирования дат, времени или того и другого с учетом заданных стилей.

- **DateTimeFormatter withLocale(Locale locale)**

Возвращает копию данного средства форматирования вместе с заданными региональными настройками.

- **String format(TemporalAccessor temporal)**

Возвращает символьную строку, получающуюся в результате форматирования заданных даты и времени.

```
java.time.LocalDate 8
```

```
java.time.LocalTime 8
```

```
java.time.LocalDateTime 8
```

```
java.time.ZonedDateTime 8
```

- **static Xxx parse(CharSequence text, DateTimeFormatter formatter)**

Производит синтаксический анализ заданной символьной строки и возвращает описанную в ней местную дату или время в виде объекта типа **LocalDate**, **LocalTime**, **LocalDateTime** или **ZonedDateTime**. Генерирует исключение типа **DateTimeParseException** при неудачном исходе синтаксического анализа.

7.5. Сортировка и нормализация

Как известно, для сравнения символьных строк служит метод `compareTo()` из класса `String`. К сожалению, этот метод не совсем годится для взаимодействия с пользователями. В методе `compareTo()` применяются строковые значения в кодировке UTF-16, что приводит к абсурдным результатам, даже на английском языке. Например, следующие пять символьных строк упорядочиваются по результатам сортировки методом `compareTo()` таким образом:

Athens
Zulu
able
zebra
Ångström

При упорядочении словаря приходится учитывать регистр букв, но совсем не обязательно ударение. Для англоязычного пользователя приведенный выше перечень слов должен быть упорядочен следующим образом:

```
able
Ångström
Athens
zebra
Zulu
```

Но такой порядок следования слов неприемлем для шведскоязычного пользователя. Ведь в шведском языке буква **Å** отличается от буквы **A**, и поэтому сортируется *после* буквы **Z**! Это означает, что для шведскоязычного пользователя упомянутый выше перечень слов должен быть отсортирован следующим образом:

```
able
Athens
zebra
Zulu
Ångström
```

Чтобы получить компаратор с учетом региональных настроек, следует вызвать метод `Collator.getInstance()`, как показано ниже. Класс `Collator` реализует интерфейс `Comparator`, и поэтому объект типа `Collator` можно передать методу `List.sort(Comparator)`, чтобы отсортировать символьные строки.

```
Collator coll = Collator.getInstance(locale);
words.sort(coll);
```

// Класс `Collator` реализует интерфейс `Comparator<Object>`

Для средств сортировки предусмотрены четыре уровня избирательности: *первостепенный, второстепенный, третьюстепенный и идентичный*. Например, в английском языке отличие букв **A** и **Z** считается первостепенным, букв **A** и **Å** — второстепенным, а букв **A** и **a** — третьестепенным.

Для того чтобы при сортировке внимание обращалось только на первостепенные отличия, следует задать уровень ее избирательности `Collator.PRIMARY`. А если задать уровень избирательности `Collator.SECONDARY`, то будут учтены и второстепенные отличия. Таким образом, вероятность найти отличия в двух символьных строках будет больше при установке более высокого уровня избирательности (табл. 7.6).

Таблица 7.6. Сортировка с разными уровнями избирательности
[английские региональные настройки]

Первостепенный уровень	Второстепенный уровень	Третьюстепенный уровень
<code>Angstrom = Ångström</code>	<code>Angstrom ≠ Ångström</code>	<code>Angstrom ≠ Ångström</code>
<code>Able = able</code>	<code>Able = able</code>	<code>Able ≠ able</code>

Если же установлен уровень избирательности `Collator.IDENTICAL`, то отличия не допускаются. Этот уровень избирательности используется главным образом вместе с режимом разложения на составляющие, который устанавливается для сортировки и рассматривается ниже.

Иногда символ или последовательность символов могут быть описаны не только в Юникоде. Например, символу **Å** в Юникоде соответствует код `U+00C5`. С другой стороны, его можно представить в виде последовательности символов **A** (код `U+0065`) и **°** (кружок сверху; код `U+030A`). Еще удивительнее, что последовательность символов "ffi" может быть описана одним символом "латинская

малая лигатура **ffi**" с кодом U+FB03. (Можно, конечно, спорить, что это вопрос представления символов, решение которого не должно приводить к появлению разных символов в Юникоде, но правила установлены не нами.)

В стандарте на Юникод определяются четыре формы нормализации символьных строк (D, KD, C и KC; подробнее об этом см. по адресу <http://www.unicode.org/unicode/reports/tr15/tr15-23.html>). Две из этих форм используются для сортировки. В форме нормализации D символы с ударением раскладываются на составляющие их буквы и ударения. Например, символ **À** раскладывается на составляющие символы **A** и **°**. А в формах нормализации KC и KD на составляющие раскладываются такие символы, как лигатура **ffi** или знак торговой марки **™**.

Для сортировки можно выбрать определенную степень нормализации. Так, если установить значение константы `Collator.NO_DECOMPOSITION`, то символьные строки вообще не будут нормализованы при сортировке. В этом режиме сортировка выполняется быстрее, но он может быть непригодным для сортировки текста, где символы выражаются во многих формах. По умолчанию устанавливается значение константы `Collator.CANONICAL_DECOMPOSITION`, определяющее режим, в котором используется форма нормализации D. Это самая полезная форма для сортировки текста, содержащего символы с ударениями, но не лигатуры. И, наконец, в режиме полного разложения на составляющие используется форма нормализации KD. Характерные примеры сортировки в режимах разложения на составляющие приведены в табл. 7.7.

Таблица 7.7. Сортировка в разных режимах разложения на составляющие

Без разложения на составляющие	Каноническое разложение на составляющие	Полное разложение на составляющие
À ≠ A	À = A	À = A
™ ≠ TM	™ ≠ TM	™ = TM

Если одна символьная строка сравнивается многократно с другими строками, то во избежание ее повторного разложения на составляющие и ради повышения эффективности результат разложения на составляющие следует сохранить в объекте ключа *сортировки*. Например, в приведенном ниже фрагменте кода метод `getCollationKey()` возвращает объект типа `CollationKey`, используемый для ускорения всех последующих операций сравнения.

```
String a = . . .;
CollationKey aKey = coll.getCollationKey(a);
if(aKey.compareTo(coll.getCollationKey(b)) == 0) // быстрое сравнение
    . . .
```

И наконец, символьные строки иногда требуется преобразовать в их нормализованные формы, не прибегая к сортировке. Такая потребность возникает, например, при сохранении символьных строк в базе данных или при взаимодействии с другой программой. Для этой цели служит класс `java.text.Normalizer`, выполняющий процесс нормализации, как показано ниже. Нормализованная строка содержит десять символов. Символы **À** и **ö** заменяются последовательностями символов "**A°**" и "**o°**".

```

String name = "Ångström";
String normalized =
    Normalizer.normalize(name, Normalizer.Form.NFD);
// использовать форму нормализации D

```

Тем не менее это обычно не самая лучшая форма для хранения и передачи символьных строк. В форме нормализации С сначала выполняется разложение на составляющие, а затем присоединяются ударения в установленном порядке. В соответствии с рекомендациями консорциума W3C такой режим является наиболее предпочтительным для передачи данных через Интернет.

Программа, исходный код которой приведен в листинге 7.4, позволяет экспериментировать с разными видами сортировки. Достаточно ввести слово в текстовом поле и щелкнуть на кнопке Add, чтобы добавить введенное слово в список. Список сортируется заново после добавления в него каждого слова, изменения региональных настроек (в раскрывающемся списке Locale), уровня избирательности сортировки (в раскрывающемся списке Strength) или режима разложения на составляющие (в раскрывающемся списке Decomposition). Знак равенства (=) обозначает, что слова считаются одинаковыми (рис. 7.3).

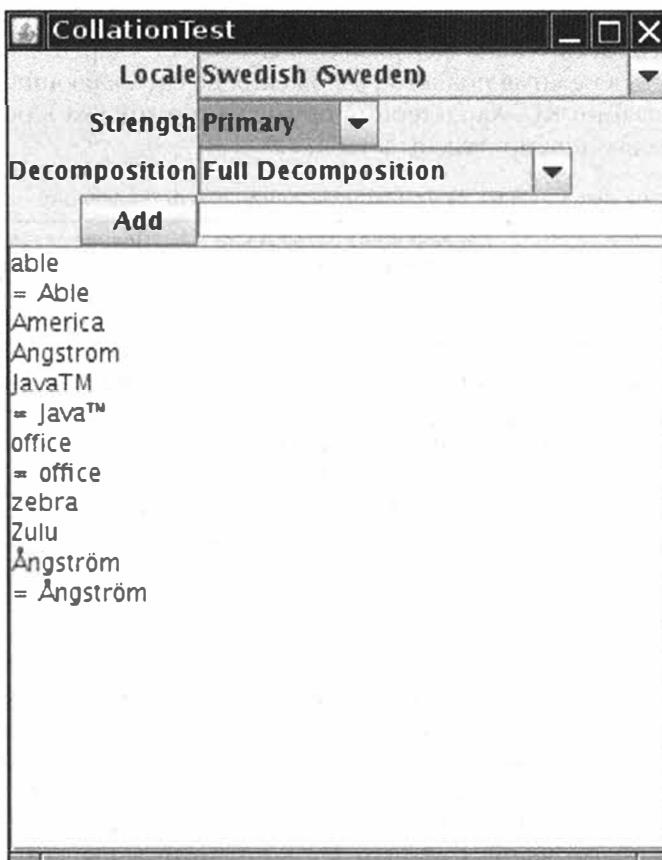


Рис. 7.3. Рабочее окно программы CollationTest

Наименования региональных настроек в раскрывающемся списке Locale отображаются в порядке, отсортированном в соответствии с устанавливаемыми по умолчанию региональными настройками. Так, если запустить рассматриваемую здесь программу при стандартных региональных настройках US English, то региональные настройки Norwegian (Norway, Nynorsk) окажутся выше в данном списке, чем региональные настройки Norwegian (Norway), несмотря на то, что значение знака запятой в Юникоде больше, чем значение знака закрывающей скобки.

Листинг 7.4. Исходный код из файла collation/CollationTest.java

```
1 package collation;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.text.*;
6 import java.util.*;
7 import java.util.List;
8
9 import javax.swing.*;
10
11 /**
12 * В этой программе демонстрируется сортировка символьных
13 * строк при выборе разных региональных настроек
14 * @version 1.15 2016-05-06
15 * @author Cay Horstmann
16 */
17 public class CollationTest
18 {
19     public static void main(String[] args)
20     {
21         EventQueue.invokeLater(() ->
22         {
23             JFrame frame = new CollationFrame();
24             frame.setTitle("CollationTest");
25             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26             frame.setVisible(true);
27         });
28     }
29 }
30
31 /**
32 * Этот фрейм содержит комбинированные списки для выбора
33 * региональных настроек, уровня избирательности сортировки и
34 * режимов разложения на составляющие, текстовое поле и кнопку
35 * для ввода новых символьных строк, а также текстовую область
36 * для перечисления отсортированных символьных строк
37 */
38
39 class CollationFrame extends JFrame
40 {
41     private Collator collator = Collator.getInstance(
42             Locale.getDefault());
43     private List<String> strings = new ArrayList<>();
44     private Collator currentCollator;
45     private Locale[] locales;
46     private JComboBox<String> localeCombo = new JComboBox<>();
47     private JTextField newWord = new JTextField(20);
48     private JTextArea sortedWords = new JTextArea(20, 20);
49     private JButton addButton = new JButton("Add");
```

```
50     private EnumCombo<Integer> strengthCombo =
51         new EnumCombo<>(Collator.class, "Primary",
52                             "Secondary", "Tertiary", "Identical");
53     private EnumCombo<Integer> decompositionCombo =
54         new EnumCombo<>(Collator.class,
55                             "Canonical Decomposition",
56                             "Full Decomposition", "No Decomposition");
57
58     public CollationFrame()
59     {
60         setLayout(new GridBagLayout());
61         add(new JLabel("Locale"), new GBC(0, 0)
62             .setAnchor(GBC.EAST));
63         add(new JLabel("Strength"), new GBC(0, 1)
64             .setAnchor(GBC.EAST));
65         add(new JLabel("Decomposition"), new GBC(0, 2)
66             .setAnchor(GBC.EAST));
67         add(addButton, new GBC(0, 3).setAnchor(GBC.EAST));
68         add(localeCombo, new GBC(1, 0).setAnchor(GBC.WEST));
69         add(strengthCombo, new GBC(1, 1).setAnchor(GBC.WEST));
70         add(decompositionCombo, new GBC(1, 2).setAnchor(GBC.WEST));
71         add(newWord, new GBC(1, 3).setFill(GBC.HORIZONTAL));
72         add(new JScrollPane(sortedWords), new GBC(0, 4, 2, 1)
73             .setFill(GBC.BOTH));
74
75         locales = (Locale[]) Collator.getAvailableLocales().clone();
76         Arrays.sort(locales, (l1, l2) ->
77             collator.compare(l1.getDisplayName(),
78                             l2.getDisplayName()));
79         for (Locale loc : locales)
80             localeCombo.addItem(loc.getDisplayName());
81         localeCombo.setSelectedItem(Locale.getDefault()
82             .getDisplayName());
83
84         strings.add("America");
85         strings.add("able");
86         strings.add("Zulu");
87         strings.add("zebra");
88         strings.add("\u00C5ngstr\u00F6m");
89         strings.add("A\u030angstro\u0308m");
90         strings.add("Angstrom");
91         strings.add("Able");
92         strings.add("office");
93         strings.add("o\uFB03ce");
94         strings.add("Java\u2122");
95         strings.add("JavaTM");
96         updateDisplay();
97
98         addButton.addActionListener(event ->
99         {
100             strings.add(newWord.getText());
101             updateDisplay();
102         });
103
104         ActionListener listener = event -> updateDisplay();
105
106         localeCombo.addActionListener(listener);
107         strengthCombo.addActionListener(listener);
108         decompositionCombo.addActionListener(listener);
109         pack();
110     }
```

```

111 /**
112 * Обновляет отображаемые строки и сортирует их
113 * в соответствии с пользовательскими установками
114 */
115 public void updateDisplay()
116 {
117     Locale currentLocale = locales[localeCombo
118                             .getSelectedIndex()];
119     localeCombo.setLocale(currentLocale);
120
121     currentCollator = Collator.getInstance(currentLocale);
122     currentCollator.setStrength(strengthCombo.getValue());
123     currentCollator.setDecomposition(
124         decompositionCombo.getValue());
125
126     Collections.sort(strings, currentCollator);
127
128     sortedWords.setText("");
129     for (int i = 0; i < strings.size(); i++)
130     {
131         String s = strings.get(i);
132         if (i > 0 && currentCollator.compare(s,
133                                         strings.get(i - 1)) == 0)
134             sortedWords.append("= ");
135         sortedWords.append(s + "\n");
136     }
137 }
138 pack();
139 }
140 }
```

java.text.Collator 1.1• **static Locale[] getAvailableLocales()**

Возвращает массив объектов типа **Locale**, для которых существуют сортирующие объекты типа **Collator**.

• **static Collator getInstance()**• **static Collator getInstance(Locale l)**

Возвращают объект типа **Collator** для текущих или заданных региональных настроек.

• **int compare(String a, String b)**

Возвращает отрицательное значение, если строка **a** предшествует строке **b**; нулевое значение, если строки считаются одинаковыми; или положительное значение, если строка **b** предшествует строке **a**.

• **boolean equals(String a, String b)**

Возвращает логическое значение **true**, если строки **a** и **b** считаются одинаковыми, а иначе — логическое значение **false**.

• **void setStrength(int strength)**• **int getStrength()**

Устанавливают или получают уровень избирательности сортировки. Чем выше уровень избирательности, тем больше вероятность того, что сортируемые слова будут признаны разными. Поддерживаются следующие уровни избирательности сортировки: **Collator.PRIMARY**, **Collator.SECONDARY** и **Collator.TERTIARY**.

java.text.Collator 1.1 (окончание)

- **void setDecomposition(int decomp)**
- **int getDecompositon()**
Устанавливают или получают режим разложения на составляющие при сортировке символьных строк. Чем выше степень разложения на составляющие, тем строже выполняется сравнение сортируемых символьных строк. Поддерживаются следующие режимы разложения на составляющие: `Collator.NO_DECOMPOSITION`, `Collator.CANONICAL_DECOMPOSITION` и `Collator.FULL_DECOMPOSITION`.
- **CollationKey getCollationKey(String a)**
Возвращает ключ сортировки с разложенными на составляющие символами, чтобы быстро сравнить их по другому ключу сортировки.

java.text.CollationKey 1.1

- **int compareTo(CollationKey b)**
Возвращает отрицательное значение, если данный ключ сортировки предшествует ключу *b*; нулевое значение, если ключи одинаковы; или положительное значение, если данный ключ следует за ключом *b*.

java.text.Normalizer 6

- **static String normalize(CharSequence str, Normalizer.Form form)**
Возвращает нормализованную форму символьной строки *str*. Параметр *form* может принимать одно из следующих значений: `ND`, `NKD`, `NC` или `NKC`.

7.6. Форматирование сообщений

В состав библиотеки Java входит класс `MessageFormat` для форматирования текста, содержащего фрагменты с переменными. Этот механизм подобен формированию с помощью метода `printf()`, но он действует с учетом региональных настроек, а также форматов чисел и дат. В последующих разделах этот механизм рассматривается более подробно.

7.6.1. Форматирование чисел и дат

Ниже приведен пример типичной строки форматирования сообщений, где номера в фигурных скобках служат в качестве заполнителей для подлинных имен и значений.

"On {2}, a {0} destroyed {1} houses and caused {3} of damage."

Подставить значения переменных можно с помощью статического метода `MessageFormat.format()` с переменным числом параметров, где подстановка значений переменных может быть произведена следующим образом:

```
String msg = MessageFormat.format("On {2}, a {0} destroyed {1} houses
and caused {3} of damage.", "hurricane", 99,
new GregorianCalendar(1999, 0, 1).getTime(), 10.0E8);
```

В данном примере заполнитель {0} замещается строковым значением "hurricane", заполнитель {1} — числовым значением 99 и т.д. А в результате подстановки получается следующая текстовая строка:

```
On 1/1/99 12:00 AM, a hurricane destroyed 99 houses and
caused 100,000,000 of damage.
```

Результат для начала неплохой, но вряд ли он может устроить полностью. В частности, время 12:00 AM отображать не следует, а сумму ущерба от урагана нужно представить в денежных единицах. Это можно сделать, указав формат для некоторых переменных, как выделено ниже полужирным.

```
"On {2,date,long}, a {0} destroyed {1} houses and caused
{3,number,currency} of damage."
```

В результате очередной подстановки получается следующая строка:

```
On January 1, 1999, a hurricane destroyed 99 houses and
caused $100,000,000 of damage.
```

Обычно после заполнителя допускается задавать *тип* и *стиль*, разделяя их запятыми. Ниже перечислены допустимые типы.

```
number
time
date
choice
```

Если указан тип *number*, то допускаются следующие стили:

```
integer
currency
percent
```

В качестве стиля может быть также указан шаблон числового формата, например \$,##0. (Подробнее об этом см. в документации на класс DecimalFormat.)

Для типа *time* или *date* может быть указан один из следующих стилей:

```
short
medium
long
full
```

Аналогично числам, в качестве стиля может быть указан шаблон даты, например, ггг-мм-дд. (Допустимые форматы подробно рассматриваются в документации на класс SimpleDateFormat.)



Внимание! Статический метод **format()** форматирует значения с учетом текущих региональных настроек. Форматировать сообщения средствами класса **MessageFormat** с учетом произвольных региональных настроек немного сложнее, поскольку в этом классе отсутствует метод с переменным числом аргументов. Поэтому форматируемые значения придется разместить в массиве **Object[]**, как показано ниже.

```
MessageFormat mf = new MessageFormat(pattern, loc);
String msg = mf.format(new Object[] { значения });
```

java.text.MessageFormat 1.1

- **MessageFormat(String pattern)**
- **MessageFormat(String pattern, Locale loc)**
- Создают объект, форматирующий сообщения по заданному шаблону и указанным региональным настройкам.
- **void applyPattern(String pattern)**
- Задает шаблон для объекта, форматирующего сообщения.
- **void setLocale(Locale loc)**
- **Locale getLocale()**
- Устанавливают или получают региональные настройки для заполнителей в сообщении. Региональные настройки пригодны только для последующих шаблонов, задаваемых с помощью метода **applyPattern()**.
- **static String format(String pattern, Object... args)**
- Форматирует символьную строку по шаблону **pattern**, подставляя вместо заполнителей **{i}** объекты из массива **args[i]**.
- **StringBuffer format(Object args, StringBuffer result, FieldPosition pos)**

Форматирует шаблон данного объекта типа **MessageFormat**. Параметр **args** принимает массив объектов. Форматируемая строка добавляется к значению параметра **result**, которое затем возвращается. Если параметр **pos** принимает ссылку на новый объект **new FieldPosition(MessageFormat.Field.ARGUMENT)**, его свойства **beginIndex** и **endIndex** устанавливаются в соответствии с расположением текста, который подставляется вместо заполнителя **{1}**. Если же сведения о расположении подстановочного текста не важны, в качестве параметра **pos** следует задать пустое значение **null**.

java.text.Format 1.1

- **String format(Object obj)**

Форматирует заданный объект по правилам, определяемым текущим форматирующими объектом. С этой целью делается следующий вызов **format(obj, new StringBuffer(), new FieldPosition(1)).toString()**.

7.6.2. Форматы выбора

Вернемся к следующему шаблону из предыдущего раздела, чтобы рассмотреть его подробнее:

"On {2}, a {0} destroyed {1} houses and caused {3} of damage."

Если вместо заполнителя **{0}**, обозначающего вид стихийного бедствия, подставить строковое значение "**earthquake**" (землетрясение), то получится следующее предложение, нарушающее правила английской грамматики в отношении используемых артиклей:

On January 1, 1999, a earthquake destroyed . . .

Для устранения этой грамматической ошибки артикль **a** придется ввести в заполнитель {0} следующим образом:

"On {2}, {0} destroyed {1} houses and caused {3} of damage."

Теперь вместо заполнителя {0} будет подставлен текст "**a hurricane**" или "**an earthquake**". Такой способ особенно удобен для перевода сообщений на языки, где в каждом роде употребляется отдельный артикль. Например, на немецком языке этот шаблон должен выглядеть так:

"{0} zerstörte am {2} {1} Häuser und richtete einen Schaden von {3} an."

В этом случае заполнитель будет заменяться грамматически правильными сочетаниями артикля и имени существительного, например "**Ein Wirbelsturm**" и "**Eine Naturkatastrophe**".

Теперь рассмотрим заполнитель {1}. Если стихийное бедствие оказалось не очень разрушительным, то вместо этого заполнителя можно подставить значение 1. Но и в этом случае получится предложение с нарушением правил английской грамматики:

On January 1, 1999, a mudslide destroyed 1 houses and . . .

Желательно, чтобы текст сообщения грамотно изменялся в соответствии с одним из следующих подставляемых значений:

no houses
one house
2 houses
. . .

Именно для этой цели и был внедрен формат выбора типа choice. В соответствии с этим форматом задается последовательность пар значений, каждая из которых содержит **нижний предел и форматирующую строку**. Нижний предел и форматирующая строка разделяются знаком #, а для разделения пар значений служит знак |. Ниже приведен пример заполнителя {1}, где формат выбора выделен полужирным. А результаты форматирования текста сообщения в зависимости от значения, подставляемого вместо заполнителя {1}, представлены в табл. 7.8.

{1, choice, 0#no houses|1#one house|2#{1} houses}

Таблица 7.8. Текст сообщения, отформатированный по выбору

{1}	Результат
0	"no houses"
1	"one house"
3	"3 houses"
-1	"no houses"

А зачем в форматирующей строке дважды указывается заполнитель {1}? Когда к этому заполнителю применяется формат выбора и значение оказывается равным 2, возвращается символьная строка "{1} houses". Эта строка форматируется еще раз и включается в результирующую строку сообщения.



На заметку! Приведенный выше пример показывает, что разработчики формата выбора приняли не самое лучшее решение. Так, если имеются три форматирующие строки, то для их разделения требуются два предела. Как правило, количество пределов должно быть на единицу меньше, чем количество форматирующих строк. И как следует из табл. 7.8, первый предел в классе `MessageFormat` вообще игнорируется. Синтаксис форматирования сообщений мог бы быть более понятным, если бы пределы указывались между выбираемыми вариантами, например, следующим образом:

```
по houses|1|one house|2|{1} houses
// более понятный, но не действующий формат
```

С помощью знака < можно указать, что предлагаемый вариант должен быть выбран, если нижний предел оказывается строго меньше подставляемого значения. Вместо знака # можно также указывать знак ≤ (\u2264 в Юникоде). По желанию можно даже указать для первого значения нижний предел равным -∞ (-\u221e в Юникоде):

```
-∞<по houses|0<one house|2≤{1} houses
```

А непосредственно в Юникоде это же выражение будет выглядеть следующим образом:

```
-\u221e<по houses|0<one house|2\u2264{1} houses
```

В завершение примера форматирования текстового сообщения о последствиях стихийного бедствия разместим строку с условиями выбора в исходной строке сообщения. В результате получится следующий шаблон форматирования на английском языке:

```
String pattern = "On {2,date,long},
{0} destroyed {1,choice,0#no houses|1#one house|2#{1} houses}"
+ "and caused {3,number,currency} of damage.";
```

А для форматирования на немецком языке этот шаблон будет выглядеть следующим образом:

```
String pattern = "{0} zerstörte am {2,date,long}
{1,choice,0#kein Haus|1#ein Haus|2#{1} Häuser}"
+ "und richtete einen Schaden von {3,number,currency} an.";
```

Примечательно, что порядок слов в шаблонах форматирования на английском и немецком языках разный, но методу `format()` передается *тот же самый* массив объектов. Под требуемый порядок слов в форматирующющей строке подстраивается только последовательность заполнителей.

7.7. Ввод-вывод текста

Как вам должно быть уже известно, кодирование символов в Java основывается на Юникоде. Но в операционных системах Windows и Mac OS X до сих пор применяются устаревшие кодировки символов, часто несовместимые с другими, например, Windows-1252 и Mac Roman в странах Западной Европы или BIG5 на Тайване. Поэтому организовать взаимодействие с пользователями через текст оказывается не так просто, как может показаться на первый взгляд. Трудности, возникающие на этом пути, рассматриваются в последующих разделах.

7.7.1. Текстовые файлы

В настоящее время для сохранения или загрузки текстовых файлов лучше всего пользоваться кодировкой UTF-8, хотя может возникнуть потребность в обработке текстовых файлов с устаревшей кодировкой. Если кодировка символов заранее известна, ее можно указать при записи или чтении текстовых файлов, как показано ниже.

```
PrintWriter out = new PrintWriter(filename, "Windows-1252");
```

Чтобы выяснить наиболее подходящую кодировку, можно получить “платформенную” кодировку, сделав следующий вызов:

```
Charset platformEncoding = Charset.defaultCharset();
```

7.7.2. Окончания строк

Правильная интерпретация окончаний строк — дело не региональных настроек, а платформ. Так, в Windows предполагается обнаружить последовательность символов `\r\n` в конце каждой строки текстового файла, тогда как в UNIX-подобных системах в конце строки достаточно указать последовательность символов `\n`. Впрочем, большинство современных прикладных программ для Windows способно правильно интерпретировать и последовательность символов `\n`. Примечательным исключением из этого правила служит текстовый редактор Notepad, и если текстовых файл, выбираемый двойным щелчком кнопкой мыши на его имени, требуется автоматически загрузить в текстовый редактор Notepad, следует обеспечить правильное окончание строк в нем.

Любая строка, выводимая с помощью метода `println()`, получает надлежащее окончание. Единственное затруднение возникает при выводе символьных строк, оканчивающихся последовательностью символов `\n`, поскольку они не преобразуются автоматически в окончания строк, принятые на конкретной платформе.

Вместо употребления последовательности символов `\n` для окончаний строк можно вызвать метод `printf()`, указав спецификатор формата `%n` для получения платформенно-ориентированных окончаний строк. Например, в результате следующего вызова:

```
out.printf("Hello%nWorld%n");
```

в Windows получается такая строка:

```
Hello\r\nWorld\r\n
```

а в других операционных системах — строка, приведенная ниже.

```
Hello\nWorld\n
```

7.7.3. Консольный ввод-вывод

При написании прикладных программ, взаимодействующих с пользователем через стандартный ввод-вывод (объекты `System.in/System.out`) или консоль (метод `System.console()`), приходится принимать во внимание, что на консоли может использоваться иная, чем платформенная, кодировка, о которой сообщает метод `Charset.defaultCharset()`. Эта особенность заметна при переходе

к командной оболочке cmd в Windows. В версии этой оболочки для США применяется архаичная кодировка IBM437, внедренная на ПК IBM еще в 1982 году. Для обнаружения подобной информации отсутствует официально утвержденный прикладной программный интерфейс API. Например, знак денежной единицы евро (€) имеет свое представление в кодировке Windows-1252, тогда как в кодировке IBM437 такое представление отсутствует. Поэтому в результате следующего вызова:

```
System.out.println("100 €");
```

на консоль выводится такой результат:

```
100 ?
```

Пользователям прикладной программы можно порекомендовать сменить кодировку символов на консоли. В Windows для этого можно воспользоваться командой chcp. Например, по следующей команде:

```
chcp 1252
```

консоль перейдет к кодовой странице Windows-1252.

В идеальном случае пользователи должны перевести консоль на кодировку UTF-8. С этой целью в Windows можно ввести следующую команду:

```
chcp 65001
```

К сожалению, этого оказывается недостаточно, чтобы на консоли в Java применялась кодировка UTF-8. Ее необходимо также установить неофициально в системном свойстве file.encoding по команде

```
java -Dfile.encoding=UTF-8 MyProg
```

7.7.4. Протокольные файлы

Когда протокольные сообщения направляются из библиотеки java.util.logging на консоль, они выводятся в кодировке, принятой на консоли. О том, как управлять этим процессом, упоминалось в предыдущем разделе. Но для вывода протокольных сообщений в файл служит класс FileHandler, где по умолчанию применяется платформенная кодировка.

Чтобы перейти к кодировке UTF-8, необходимо внести изменения в настройки диспетчера протоколирования. С этой целью в файле конфигурации протоколирования делается следующая установка:

```
java.util.logging.FileHandler.encoding=UTF-8
```

7.7.5. Отметка порядка следования байтов в кодировке UTF-8

Как упоминалось ранее, для обработки текстовых файлов рекомендуется применять кодировку UTF-8 при всякой возможности. Так, при чтении в прикладной программе текстовых файлов, созданных в других программах, может возникнуть еще одно затруднение в связи с тем, что в файл вполне допускается вводить символ, обозначающий отметку порядка следования байтов (U+FEFF). В кодировке UTF-16, где каждая кодовая единица представлена двумя байтами, такая отметка сообщает программе, читающей файл, что в нем применяется порядок следования байтов от старшего к младшему или же от младшего

к старшему. А в кодировке UTF-8 каждая кодовая единица представлена одним байтом, и поэтому указывать порядок следования байтов в этом случае не требуется. Но если файл начинается с байтов **0xEF 0xBB 0xBF**, что соответствует коду символа **U+FEFF** в кодировке UTF-8, то это явно свидетельствует о применении кодировки UTF-8. Именно такая практика и рекомендуется в стандарте Unicode, когда любая программа, читающая файл, отбрасывает первоначальную отметку порядка следования байтов.

Такой практике присущ лишь один недостаток. Компания Oracle в своей реализации языка Java упорно отказывается следовать стандарту Unicode, ссылаясь на возможную потенциальную несовместимость. А это означает, что программирующие на Java должны сделать то, что не сделает платформа, а именно: проигнорировать код символа **U+FEFF**, если он встретится в начале читаемого текстового файла.



Внимание! К сожалению, разработчики комплекта JDK не следуют приведенной выше рекомендации. Если передать компилятору **javac** достоверный исходный файл в кодировке UTF-8, который начинается с отметки порядка следования байтов, его компиляция завершится неудачно выдачей сообщения об ошибке "**illegal character: \65279**" (недопустимый символ: код \65279).

7.7.6. Кодирование символов в исходных файлах

Не следует забывать, что при написании программы на Java неизбежно приходится иметь дело с компилятором, пользуясь инструментальными средствами в локальной операционной системе. Допустим, что для создания исходных файлов программы на Java используется стандартный текстовый редактор Notepad в китайской версии Windows. Полученный в итоге исходный код не является переносимым (т.е. независимым от платформы) из-за того, что в нем используется локальная кодировка символов (GB или BIG5, в зависимости от региональных настроек операционной системы). Классы становятся переносимыми только после компиляции, и в этом случае идентификаторы и текстовые сообщения представляются с помощью модифицированной кодировки UTF-8. А это означает, что при компиляции и выполнении программы предполагается использование трех перечисленных ниже кодировок символов.

- Исходные файлы: локальная кодировка.
- Файлы классов: модифицированная кодировка UTF-8.
- Виртуальная машина: кодировка UTF-16.

(Модифицированная кодировка UTF-8 и кодировка UTF-16 описаны в главе 2.)



Совет! Конкретную кодировку исходного файла можно указать в командной строке с помощью параметра **-encoding**:

```
javac -encoding Big5 Myfile.java
```

Чтобы сделать исходные файлы переносимыми, следует ограничиться только символами в коде ASCII. Иными словами, все символы, отсутствующие в коде ASCII, необходимо заменить их эквивалентными представлением в Юникоде.

Например, вместо символьной строки "Häuser" следует использовать символьную строку "H\u00f6user". Для преобразования текстов, набранных в локальной кодировке, в символы кода ASCII можно воспользоваться утилитой native2ascii, входящей в состав JDK. Эта утилита заменяет каждый символ, отсутствующий в коде ASCII, кодовой последовательностью \u~~xxxx~~хххх, где хххх – четыре шестнадцатеричные цифры кода данного символа в Юникоде. Применяя утилиту native2ascii, в командной строке необходимо указать имя исходного и целевого файлов следующим образом:

```
native2ascii Myfile.java Myfile.temp
```

А для обратного преобразования служит параметр `-reverse`, как показано ниже.

```
native2ascii -reverse Myfile.temp Myfile.java
```

И наконец, для указания другой кодировки служит параметр `-encoding`, после которого в командной строке следует название требующейся кодировки, как показано в приведенном ниже примере.

```
native2ascii -encoding Big5 Myfile.java Myfile.temp
```

7.8. Комплекты ресурсов

При интернационализации приложений на другие языки приходится переводить огромное количество сообщений, надписей на кнопках и т.п. Для упрощения этой задачи рекомендуется собрать все переводимые символьные строки в отдельном месте, которое обычно называется *ресурсом*. В этом случае переводчику достаточно отредактировать файлы ресурсов, не затрагивая исходный код программы. В языке Java для определения строковых ресурсов используются файлы свойств, а для других разновидностей ресурсов создаются классы ресурсов.



На заметку! Технология использования ресурсов в Java отличается от аналогичной технологии в операционных системах Windows и Mac OS. В программе, предназначенному для выполнения под Windows или Mac OS, такие ресурсы, как меню, диалоговые окна, пиктограммы и сообщения, хранятся отдельно от самой программы. Поэтому специальный редактор ресурсов позволяет просматривать и видоизменять эти ресурсы, не затрагивая программный код.



На заметку! В главе 13 первого тома настоящего издания описывается принцип размещения ресурсов [файлов данных, звука и изображения] в архивном JAR-файле. Метод `getResource()` из класса `Class` находит нужный файл, открывает его и возвращает URL, указывающий на искомый ресурс. При размещении файлов в архивном JAR-файле поиск файлов возлагается на загрузчик классов. Однако этот механизм не поддерживает региональные настройки.

7.8.1. Обнаружение комплектов ресурсов

Для интернационализации приложений создаются так называемые *комплекты ресурсов*. Каждый комплект представляет собой файл свойств или класс, который описывает элементы, специфические для конкретных региональных

настроек, например, сообщения, надписи и т.д. Для каждого комплекта ресурсов должны быть предоставлены все региональные настройки, поддержка которых предусматривается в прикладной программе.

Комплекты ресурсов именуются по специальным условным обозначениям. Например, ресурсы, специфические для Германии, размещаются в файле `имяКомплекта_de_DE`, а ресурсы, общие для стран, в которых используется немецкий язык, — в классе `имяКомплекта_de`. В целом комплекты ресурсов для отдельных стран именуются следующим образом:

`имяКомплекта_язык_страна`

А комплекты ресурсов для всех одноязычных стран именуются таким образом:
`имяКомплекта_язык`

И наконец, в качестве резерва ресурсы, применяемые по умолчанию, размещаются в файле, имя которого не указывается без упомянутых выше суффиксов. А загружается комплект ресурсов следующим образом:

```
ResourceBundle currentResources =  
    ResourceBundle.getBundle(имяКомплекта,  
        текущиеРегиональныеНастройки);
```

Метод `getBundle()` пытается загрузить комплект ресурсов, совпадающий с текущими региональными настройками по языку и стране. Если попытка загрузки завершится неудачей, то поочередно опускается страна и язык. Затем аналогичный поиск ресурсов осуществляется в региональных настройках по умолчанию, и, наконец, происходит обращение к комплекту ресурсов, выбранному по умолчанию. Если и эта попытка оказывается безуспешной, то генерируется исключение типа `MissingResourceException`.

Таким образом, метод `getBundle()` пытается загрузить комплекты ресурсов в приведенной ниже последовательности, где *TPH* — текущие региональные настройки, а *RNU* — региональные настройки по умолчанию.

```
имяКомплекта_языкTPH_странаTPH  
имяКомплекта_языкTPH  
имяКомплекта_языкTPH_странаRNU  
имяКомплекта_языкRNU  
имяКомплекта
```

Даже после того, как метод `getBundle()` обнаружит комплект ресурсов, например `имяКомплекта_de_DE`, он продолжит поиск ресурсов в комплектах `имяКомплекта_de` и `имяКомплекта`. Если эти комплекты ресурсов существуют, они становятся родительскими по отношению к комплекту `имяКомплекта_de_DE` в иерархии ресурсов. Родительские комплекты требуются на тот случай, если нужный ресурс не удастся обнаружить в текущем комплекте. Иными словами, если нужный ресурс не найден в комплекте `имяКомплекта_de_DE`, его поиск продолжается в комплектах в `имяКомплекта_de` и `имяКомплекта`.

Очевидно, что это очень полезный механизм, но для его реализации пришлось бы немало программировать вручную. Поэтому механизм поддержки комплектов ресурсов в Java автоматически находит ресурсы, в наибольшей степени соответствующие конкретным региональным настройкам. Чтобы ввести в существующую программу новые региональные настройки, достаточно дополнить ими соответствующие комплекты ресурсов.



На заметку! Обнаружение комплектов ресурсов рассматривается здесь в упрощенном виде. Если региональные настройки содержат письмо или вариант языка, поиск ресурсов значительно усложняется. Подробнее об этом можно узнать из документации на метод `ResourceBundle.Control.getCandidateLocales()`.



Совет! При разработке прикладной программы совсем не обязательно размещать все ресурсы в одном комплекте. Вместо этого один комплект ресурсов можно создать для надписей на кнопках, другой — для сообщений об ошибках и т.д.

7.8.2. Файлы свойств

Интернационализация символьных строк осуществляется довольно просто. Для этого достаточно разместить все символьные строки в файле свойств, например `MyProgramStrings.properties`. Файл свойств представляет собой обычный текстовый файл, каждая строка которого содержит ключ и значение, как показано ниже.

```
computeButton=Rechnen  
colorName=black  
defaultPaperSize=210x297
```

Файлы свойств именуются по принципу, описанному в предыдущем разделе, например, следующим образом:

```
MyProgramStrings.properties  
MyProgramStrings_en.properties  
MyProgramStrings_de_DE.properties
```

Комплект ресурсов можно загрузить аналогично приведенному ниже.

```
 ResourceBundle bundle =  
     ResourceBundle.getBundle("MyProgramStrings", locale);
```

А для поиска конкретной символьной строки потребуется вызов, подобный следующему:

```
String computeButtonLabel = bundle.getString("computeButton");
```



Внимание! Файлы свойств могут содержать только символы в коде ASCII. Для размещения в них символов в Юникоде следует использовать формат \uxxxx. Например, строка `colorName=Grün` будет иметь следующий вид:

```
colorName=Gr\u00FCn
```

Для подобного преобразования символов в файлах свойств можно также воспользоваться упоминавшейся ранее утилитой `native2ascii`.

7.8.3. Классы комплектов ресурсов

Для поддержки ресурсов, не являющихся символьными строками, следует определить классы, производные от класса `ResourceBundle`. Выбор имен для таких классов осуществляется в соответствии со стандартными обозначениями, как показано в приведенном ниже примере.

```
MyProgramResources.java
MyProgramResources_en.java
MyProgramResources_de_DE.java
```

Для загрузки класса комплекта ресурсов применяется тот же самый метод `getBundle()`, что и для загрузки файла свойств:

```
ResourceBundle bundle =
    ResourceBundle.getBundle("MyProgramResources", locale);
```



Внимание! Если два комплекта ресурсов, один из которых реализован в виде класса, а другой — в виде файла свойств, имеют одинаковые имена, то при загрузке предпочтение отдается классу.

В каждом классе комплекта ресурсов реализуется таблица поиска. Для получения каждого интернационализируемого значения следует предоставить символьную строку с соответствующим ключом, как показано ниже.

```
Color backgroundColor = (Color) bundle.getObject("backgroundColor");
double[] paperSize = (double[]) bundle.getObject("defaultPaperSize");
```

Самый простой способ реализовать класс комплекта ресурсов — создать подкласс, производный от класса `ListResourceBundle`. Класс `ListResourceBundle` позволяет разместить сначала все ресурсы в массиве объектов, а затем выполнить их поиск. Общая форма реализации подкласса, производного от класса `ListResourceBundle`, выглядит следующим образом:

```
public class bundleName_language_country extends ListResourceBundle
{
    private static final Object[][] contents =
    {
        { ключ1, значение1 },
        { ключ2, значение2 },
        .
        .
    }
    public Object[][] getContents() { return contents; }
}
```

Ниже приведены некоторые примеры реализации подклассов, производных от класса `ListResourceBundle`.

```
public class ProgramResources_de extends ListResourceBundle
{
    private static final Object[][] contents =
    {
        { "backgroundColor", Color.black },
        { "defaultPaperSize", new double[] { 210, 297 } }
    }
    public Object[][] getContents() { return contents; }
}

public class ProgramResources_en_US extends ListResourceBundle
{
    private static final Object[][] contents =
    {
        { "backgroundColor", Color.blue },
        { "defaultPaperSize", new double[] { 216, 279 } }
    }
    public Object[][] getContents() { return contents; }
}
```



На заметку! Размеры стандартных форматов бумаги задаются в миллиметрах. Во всех странах мира, кроме США и Канады, используются форматы бумаги, размеры которых определяются по стандарту ISO 216 (дополнительные сведения по данному вопросу можно найти в документе, доступном по адресу <http://www.cl.cam.ac.uk/~mgk25/iso-paper.html>).

Класс комплекта ресурсов можно также создать как подкласс, производный от класса `ResourceBundle`. В этом случае придется реализовать два приведенных ниже метода для получения объекта типа `Enumeration`, содержащего ключи, а также для извлечения конкретного значения по заданному ключу. Метод `getObject()` из класса `ResourceBundle` вызывает определяемый пользователем метод `handleGetObject()`.

```
Enumeration<String> getKeys()
Object handleGetObject(String key)
```

java.util.ResourceBundle 1.1

- **static ResourceBundle getBundle(String baseName, Locale loc)**
Загружают класс комплекта ресурсов по указанному имени, а также его родительские классы в соответствии с заданными или устанавливаемыми по умолчанию региональными настройками. Если классы комплектов ресурсов находятся в пакете, то должно быть указано полное имя такого класса, например `intl.Programresources`. Классы комплектов ресурсов должны быть объявлены открытыми (`public`), чтобы сделать их доступными для метода `getBundle()`.
- **Object getObject(String name)**
Извлекает объект из указанного комплекта ресурсов или его родительских комплектов.
- **String getString(String name)**
Извлекает объект из указанного комплекта ресурсов или его родительских комплектов и приводит его к типу `String`.
- **String[] getStringArray(String name)**
Извлекает объект из комплекта ресурсов или его родительских комплектов и представляет его в виде массива символьных строк.
- **Enumeration<String> getKeys()**
Возвращает объект перечисления типа `Enumeration`, содержащий ключи из текущего комплекта ресурсов. В этом объекте перечисляются также ключи из родительских комплектов ресурсов.
- **Object handleGetObject(String key)**
Если реализуется собственный механизм поиска ресурсов, этот метод следует переопределить таким образом, чтобы он возвращал значение по заданному ключу.

7.9. Пример интернационализации прикладной программы

В этом разделе изложенный выше материал применяется на практике для интернационализации прикладной программы калькуляции пенсионных сбережений. В этой программе определяется, достаточно ли отчисляется денежных средств для выхода на пенсию. Для этого следует ввести свой возраст, сумму, которая отчисляется каждый месяц, а также указать другие данные (рис. 7.4).

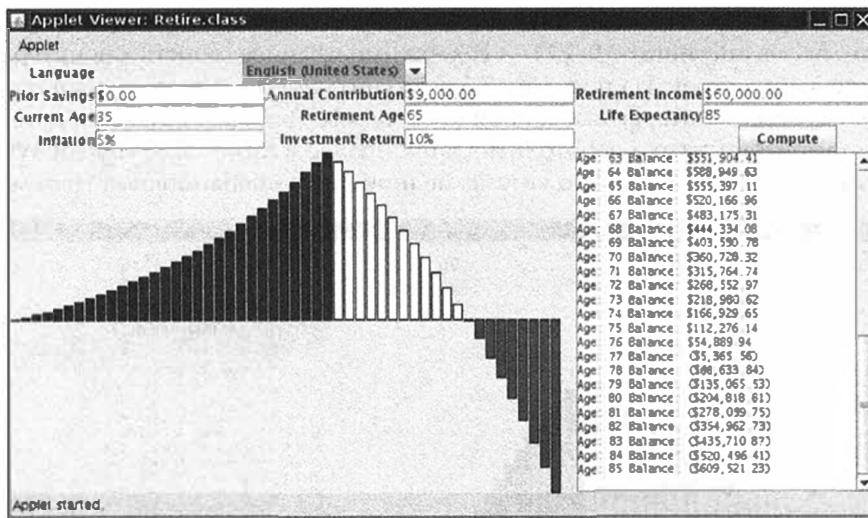


Рис. 7.4. Рабочее окно программы калькуляции пенсионных сбережений с ГПИ на английском языке

В текстовой области и на графике представлены данные о ежегодном состоянии пенсионного счета. Если с возрастом остатки на пенсионном счете становятся отрицательными, что наглядно показано на части графика, построенной ниже оси абсцисс, то следует принять какие-то меры, например, строже экономить деньги, отложить дату выхода на пенсию и т.п.

Программа калькуляции пенсионных сбережений имеет три варианта пользовательского интерфейса на английском, немецком и китайском языках. Ниже перечислены основные особенности интернационализации данной прикладной программы.

- Надписи и сообщения переводятся с английского на немецкий и китайский языки. Соответствующие ресурсы находятся в классах `RetireResources_de` и `RetireResources_zh`, а ресурсы для пользовательского интерфейса на английском языке используются в качестве резервных и находятся в классе `RetireResources`. Для составления сообщений на китайском языке был использован текстовый редактор Notepad в китайской версии Windows, а затем с помощью утилиты `native2ascii` текст был преобразован в Юникод.
- При изменении региональных настроек заново форматируются надписи и содержимое текстовых полей.
- Содержимое текстовых полей для ввода чисел, денежных сумм и процентов представляется в формате, соответствующем выбранным региональным настройкам.
- Для форматирования расчетного поля используется класс `MessageFormat`. Форматирующая строка хранится в комплекте ресурсов для каждого языка.
- Цвет графика меняется в зависимости от выбранного языка. Это никак не влияет на выполнение программой ее функций, а лишь демонстрирует подобную возможность.

В листингах 7.5–7.8 приведен исходный код рассматриваемой здесь прикладной программы, а в листингах 7.9–7.11 — содержимое файлов свойств для интернационализированных строк. На рис. 7.5 и 7.6 показаны немецкий и китайский варианты пользовательского интерфейса данной программы. Для работы с данной программой на китайском языке необходимо запустить ее в китайской версии Windows или установить самостоятельно китайские шрифты в операционной системе.

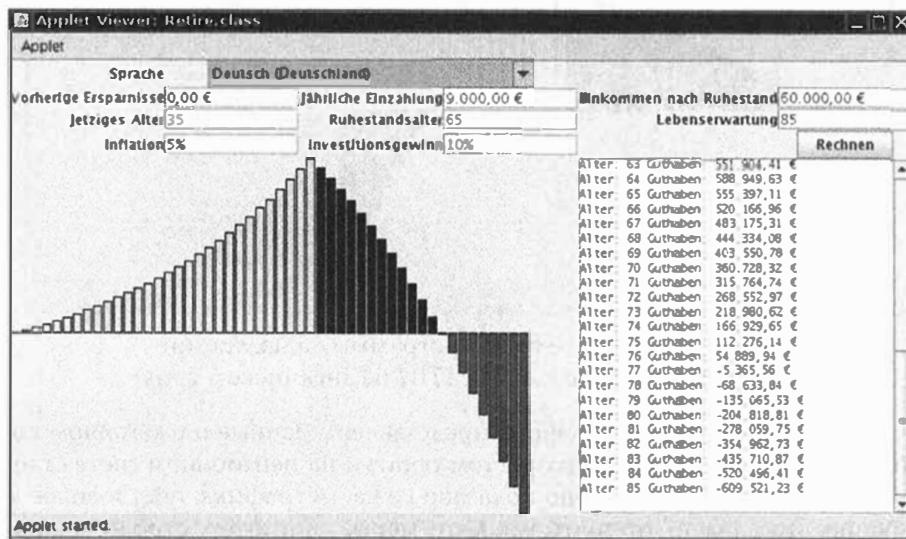


Рис. 7.5. Рабочее окно программы калькуляции пенсионных сбережений с ГПИ на немецком языке

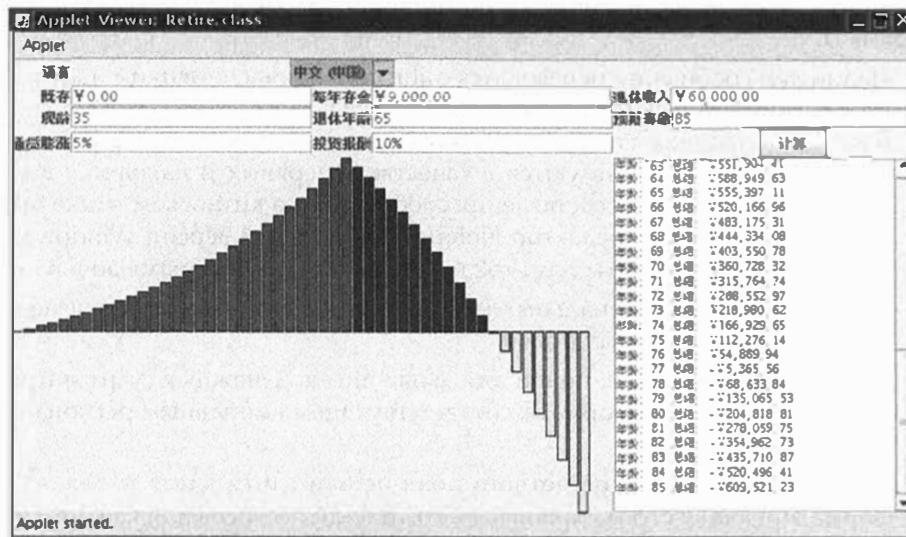


Рис. 7.6. Рабочее окно программы калькуляции пенсионных сбережений с ГПИ на китайском языке

Листинг 7.5. Исходный код из файла retire/Retire.java

```
1 package retire;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import java.text.*;
6 import java.util.*;
7
8 import javax.swing.*;
9
10 /**
11 * В этой программе демонстрируется калькуляция пенсионных
12 * сбережений. Ее пользовательский интерфейс представлен на
13 * английском, немецком и китайском языках
14 * @version 1.24 2016-05-06
15 * @author Cay Horstmann
16 */
17 public class Retire
18 {
19     public static void main(String[] args)
20     {
21         EventQueue.invokeLater(() ->
22         {
23             JFrame frame = new RetireFrame();
24             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25             frame.setVisible(true);
26         });
27     }
28 }
29
30 class RetireFrame extends JFrame
31 {
32     private JTextField savingsField = new JTextField(10);
33     private JTextField contribField = new JTextField(10);
34     private JTextField incomeField = new JTextField(10);
35     private JTextField currentAgeField = new JTextField(4);
36     private JTextField retireAgeField = new JTextField(4);
37     private JTextField deathAgeField = new JTextField(4);
38     private JTextField inflationPercentField = new JTextField(6);
39     private JTextField investPercentField = new JTextField(6);
40     private JTextArea retireText = new JTextArea(10, 25);
41     private RetireComponent retireCanvas = new RetireComponent();
42     private JButton computeButton = new JButton();
43     private JLabel languageLabel = new JLabel();
44     private JLabel savingsLabel = new JLabel();
45     private JLabel contribLabel = new JLabel();
46     private JLabel incomeLabel = new JLabel();
47     private JLabel currentAgeLabel = new JLabel();
48     private JLabel retireAgeLabel = new JLabel();
49     private JLabel deathAgeLabel = new JLabel();
50     private JLabel inflationPercentLabel = new JLabel();
51     private JLabel investPercentLabel = new JLabel();
52     private RetireInfo info = new RetireInfo();
53     private Locale[] locales = { Locale.US, Locale.CHINA,
54                                 Locale.GERMANY };
55     private Locale currentLocale;
56     private JComboBox<Locale> localeCombo =
57             new LocaleCombo(locales);
58     private ResourceBundle res;
```

```
59     private ResourceBundle resStrings;
60     private NumberFormat currencyFmt;
61     private NumberFormat numberFmt;
62     private NumberFormat percentFmt;
63
64     public RetireFrame()
65     {
66         setLayout(new GridBagLayout());
67         add(languageLabel, new GBC(0, 0).setAnchor(GBC.EAST));
68         add(savingsLabel, new GBC(0, 1).setAnchor(GBC.EAST));
69         add(contribLabel, new GBC(2, 1).setAnchor(GBC.EAST));
70         add(incomeLabel, new GBC(4, 1).setAnchor(GBC.EAST));
71         add(currentAgeLabel, new GBC(0, 2).setAnchor(GBC.EAST));
72         add(retireAgeLabel, new GBC(2, 2).setAnchor(GBC.EAST));
73         add(deathAgeLabel, new GBC(4, 2).setAnchor(GBC.EAST));
74         add(inflationPercentLabel, new GBC(0, 3)
75             .setAnchor(GBC.EAST));
76         add(investPercentLabel, new GBC(2, 3).setAnchor(GBC.EAST));
77         add(localeCombo, new GBC(1, 0, 3, 1));
78         add(savingsField, new GBC(1, 1).setWeight(100, 0)
79             .setFill(GBC.HORIZONTAL));
80         add(contribField, new GBC(3, 1).setWeight(100, 0)
81             .setFill(GBC.HORIZONTAL));
82         add(incomeField, new GBC(5, 1).setWeight(100, 0)
83             .setFill(GBC.HORIZONTAL));
84         add(currentAgeField, new GBC(1, 2).setWeight(100, 0)
85             .setFill(GBC.HORIZONTAL));
86         add(retireAgeField, new GBC(3, 2).setWeight(100, 0)
87             .setFill(GBC.HORIZONTAL));
88         add(deathAgeField, new GBC(5, 2).setWeight(100, 0)
89             .setFill(GBC.HORIZONTAL));
90         add(inflationPercentField, new GBC(1, 3)
91             .setWeight(100, 0).setFill(GBC.HORIZONTAL));
92         add(investPercentField, new GBC(3, 3)
93             .setWeight(100, 0).setFill(GBC.HORIZONTAL));
94         add(retireCanvas, new GBC(0, 4, 4, 1).setWeight(100, 100)
95             .setFill(GBC.BOTH));
96         add(new JScrollPane(retireText), new GBC(4, 4, 2, 1)
97             .setWeight(0, 100).setFill(GBC.BOTH));
98
99         computeButton.setName("computeButton");
100        computeButton.addActionListener(event ->
101        {
102            getInfo();
103            updateData();
104            updateGraph();
105        });
106        add(computeButton, new GBC(5, 3));
107
108        retireText.setEditable(false);
109        retireText.setFont(new Font("Monospaced", Font.PLAIN, 10));
110
111        info.setSavings(0);
112        info.setContrib(9000);
113        info.setIncome(60000);
114        info.setCurrentAge(35);
115        info.setRetireAge(65);
116        info.setDeathAge(85);
117        info.setInvestPercent(0.1);
118        info.setInflationPercent(0.05);
```

```
119     int localeIndex = 0; // региональные настройки США
120                     // выбираются по умолчанию
121     for (int i = 0; i < locales.length; i++)
122         // если текущие региональные настройки относятся
123         // к числу выбираемых, то выбрать их
124         if (getLocale().equals(locales[i])) localeIndex = i;
125     setCurrentLocale(locales[localeIndex]);
126
127     localeCombo.addActionListener(event ->
128     {
129         setCurrentLocale((Locale)
130                         localeCombo.getSelectedItem());
131         validate();
132     });
133     pack();
134 }
135
136 /**
137 * Устанавливает текущие региональные настройки
138 * @param locale Требующиеся региональные настройки
139 */
140 public void setCurrentLocale(Locale locale)
141 {
142     currentLocale = locale;
143     localeCombo.setLocale(currentLocale);
144     localeCombo.setSelectedItem(currentLocale);
145
146     res = ResourceBundle.getBundle(
147             "retire.RetireResources", currentLocale);
148     resStrings = ResourceBundle.getBundle(
149             "retire.RetireStrings", currentLocale);
150     currencyFmt = NumberFormat
151             .getCurrencyInstance(currentLocale);
152     numberFmt = NumberFormat.getNumberInstance(currentLocale);
153     percentFmt = NumberFormat.getPercentInstance(currentLocale);
154
155     updateDisplay();
156     updateInfo();
157     updateData();
158     updateGraph();
159 }
160
161 /**
162 * Обновляет все метки при отображении
163 */
164 public void updateDisplay()
165 {
166     languageLabel.setText(resStrings.getString("language"));
167     savingsLabel.setText(resStrings.getString("savings"));
168     contribLabel.setText(resStrings.getString("contrib"));
169     incomeLabel.setText(resStrings.getString("income"));
170     currentAgeLabel.setText(resStrings.getString("currentAge"));
171     retireAgeLabel.setText(resStrings.getString("retireAge"));
172     deathAgeLabel.setText(resStrings.getString("deathAge"));
173     inflationPercentLabel.setText(
174             resStrings.getString("inflationPercent"));
175     investPercentLabel.setText(
176             resStrings.getString("investPercent"));
177     computeButton.setText(resStrings.getString("computeButton"));
178 }
```

```
179     }
180
181     /**
182      * Обновляет данные в текстовых полях
183      */
184     public void updateInfo()
185     {
186         savingsField.setText(currencyFmt.format(info.getSavings()));
187         contribField.setText(currencyFmt.format(info.getContrib()));
188         incomeField.setText(currencyFmt.format(info.getIncome()));
189         currentAgeField.setText(numberFmt.format(
190             info.getCurrentAge()));
191         retireAgeField.setText(numberFmt.format(
192             info.getRetireAge()));
193         deathAgeField.setText(numberFmt.format(info.getDeathAge()));
194         investPercentField.setText(percentFmt.format(
195             info.getInvestPercent()));
196         inflationPercentField.setText(percentFmt.format(
197             info.getInflationPercent()));
198     }
199
200
201     /**
202      * Обновляет данные, отображаемые в текстовой области
203      */
204     public void updateData()
205     {
206         retireText.setText("");
207         MessageFormat retireMsg = new MessageFormat("");
208         retireMsg.setLocale(currentLocale);
209         retireMsg.applyPattern(resStrings.getString("retire"));
210
211         for (int i = info.getCurrentAge();
212              i <= info.getDeathAge(); i++)
213         {
214             Object[] args = { i, info.getBalance(i) };
215             retireText.append(retireMsg.format(args) + "\n");
216         }
217     }
218
219     /**
220      * Обновляет график
221      */
222     public void updateGraph()
223     {
224         retireCanvas.setColorPre((Color)
225             res.getObject("colorPre"));
226         retireCanvas.setColorGain((Color)
227             res.getObject("colorGain"));
228         retireCanvas.setColorLoss((Color)
229             res.getObject("colorLoss"));
230         retireCanvas.setInfo(info);
231         repaint();
232     }
233
234     /**
235      * Считывает данные, вводимые пользователем в текстовых полях
236      */
237     public void getInfo()
238     {
239         try
```

```
240     {
241         info.setSavings(currencyFmt.parse(savingsField.getText())
242                         .doubleValue());
243         info.setContrib(currencyFmt.parse(contribField.getText())
244                         .doubleValue());
245         info.setIncome(currencyFmt.parse(incomeField.getText())
246                         .doubleValue());
247         info.setCurrentAge(numberFmt.parse(currentAgeField
248                         .getText()).intValue());
249         info.setRetireAge(numberFmt.parse(retireAgeField
250                         .getText()).intValue());
251         info.setDeathAge(numberFmt.parse(deathAgeField.getText())
252                         .intValue());
253         info.setInvestPercent(percentFmt.parse(investPercentField
254                         .getText()).doubleValue());
255         info.setInflationPercent(percentFmt
256                         .parse(inflationPercentField.getText())
257                         .doubleValue());
258     }
259     catch (ParseException ex)
260     {
261         ex.printStackTrace();
262     }
263 }
264 }
265
266 /**
267 * Данные, требующиеся для расчета пенсионных отчислений
268 */
269 class RetireInfo
270 {
271     private double savings;
272     private double contrib;
273     private double income;
274     private int currentAge;
275     private int retireAge;
276     private int deathAge;
277     private double inflationPercent;
278     private double investPercent;
279     private int age;
280     private double balance;
281
282     /**
283      * Получает остаток на счете, имеющийся на указанный год
284      * @param year Год для расчета остатка на счете
285      * @return Возвращает сумму, имеющуюся (или требуемую)
286      *         на указанный год
287      */
288     public double getBalance(int year)
289     {
290         if (year < currentAge) return 0;
291         else if (year == currentAge)
292         {
293             age = year;
294             balance = savings;
295             return balance;
296         }
297         else if (year == age) return balance;
298         if (year != age + 1) getBalance(year - 1);
299         age = year;
```

```
300     if (age < retireAge) balance += contrib;
301     else balance -= income;
302     balance = balance * (1 + (investPercent - inflationPercent));
303     return balance;
304 }
305
306 /**
307 * Получает сумму предыдущих сбережений
308 * @return Возвращает сумму сбережений
309 */
310 public double getSavings()
311 {
312     return savings;
313 }
314
315 /**
316 * Устанавливает сумму предыдущих сбережений
317 * @param newValue Сумма сбережений
318 */
319 public void setSavings(double newValue)
320 {
321     savings = newValue;
322 }
323
324 /**
325 * Получает сумму ежегодных отчислений на пенсионный счет
326 * @return Возвращает сумму ежегодных отчислений
327 */
328 public double getContrib()
329 {
330     return contrib;
331 }
332
333 /**
334 * Устанавливает сумму ежегодных отчислений на пенсионный счет
335 * @param newValue Сумма отчислений
336 */
337 public void setContrib(double newValue)
338 {
339     contrib = newValue;
340 }
341
342 /**
343 * Получает сумму ежегодного дохода
344 * @return Возвращает сумму ежегодного дохода
345 */
346 public double getIncome()
347 {
348     return income;
349 }
350
351 /**
352 * Устанавливает сумму ежегодного дохода
353 * @param newValue Сумма ежегодного дохода
354 */
355 public void setIncome(double newValue)
356 {
357     income = newValue;
358 }
359 }
```

```
360  /**
361   * Получает текущий возраст
362   * @return Возвращает текущий возраст
363   */
364  public int getCurrentAge()
365  {
366      return currentAge;
367  }
368
369  /**
370   * Устанавливает текущий возраст
371   * @param newValue Текущий возраст
372   */
373  public void setCurrentAge(int newValue)
374  {
375      currentAge = newValue;
376  }
377
378  /**
379   * Получает возраст, требующийся для выхода на пенсию
380   * @return Возвращает пенсионный возраст
381   */
382  public int getRetireAge()
383  {
384      return retireAge;
385  }
386
387  /**
388   * Устанавливает возраст, требующийся для выхода на пенсию
389   * @param newValue Пенсионный возраст
390   */
391  public void setRetireAge(int newValue)
392  {
393      retireAge = newValue;
394  }
395
396  /**
397   * Получает предполагаемую продолжительность жизни
398   * @return Возвращает предполагаемую продолжительность жизни
399   */
400  public int getDeathAge()
401  {
402      return deathAge;
403  }
404
405  /**
406   * Устанавливает предполагаемую продолжительность жизни
407   * @param newValue Предполагаемая продолжительность жизни
408   */
409  public void setDeathAge(int newValue)
410  {
411      deathAge = newValue;
412  }
413
414  /**
415   * Получает предполагаемый уровень инфляции в процентах
416   * @return Возвращает уровень инфляции в процентах
417   */
418  public double getInflationPercent()
419  {
```

```
420     return inflationPercent;
421 }
422
423 /**
424 * Устанавливает предполагаемый уровень инфляции в процентах
425 * @param newValue Предполагаемый уровень инфляции в процентах
426 */
427 public void setInflationPercent(double newValue)
428 {
429     inflationPercent = newValue;
430 }
431
432 /**
433 * Получает предполагаемый доход от капиталовложений
434 * @return Возвращает доход от капиталовложений
435 */
436 public double getInvestPercent()
437 {
438     return investPercent;
439 }
440
441 /**
442 * Устанавливает предполагаемый доход от капиталовложений
443 * @param newValue Доход от капиталовложений в процентах
444 */
445 public void setInvestPercent(double newValue)
446 {
447     investPercent = newValue;
448 }
449 }
450
451 /**
452 * Этот компонент рисует график результатов пенсионных вложений
453 */
454 class RetireComponent extends JComponent
455 {
456     private static final int PANEL_WIDTH = 400;
457     private static final int PANEL_HEIGHT = 200;
458     private static final Dimension PREFERRED_SIZE =
459         new Dimension(800, 600);
460     private RetireInfo info = null;
461     private Color colorPre;
462     private Color colorGain;
463     private Color colorLoss;
464
465     public RetireComponent()
466     {
467         setSize(PANEL_WIDTH, PANEL_HEIGHT);
468     }
469
470 /**
471 * Устанавливает данные для построения графика
472 * пенсионных вложений
473 * @param newInfo Новые данные о пенсионных вложениях
474 */
475 public void setInfo(RetireInfo newInfo)
476 {
477     info = newInfo;
478     repaint();
479 }
```

```
480
481     public void paintComponent(Graphics g)
482     {
483         Graphics2D g2 = (Graphics2D) g;
484         if (info == null) return;
485
486         double minValue = 0;
487         double maxValue = 0;
488         int i;
489         for (i = info.getCurrentAge();
490              i <= info.getDeathAge(); i++)
491         {
492             double v = info.getBalance(i);
493             if (minValue > v) minValue = v;
494             if (maxValue < v) maxValue = v;
495         }
496         if (maxValue == minValue) return;
497
498         int barWidth = getWidth() /
499             (info.getDeathAge() - info.getCurrentAge() + 1);
500         double scale = getHeight() / (maxValue - minValue);
501
502         for (i = info.getCurrentAge();
503              i <= info.getDeathAge(); i++)
504         {
505             int x1 = (i - info.getCurrentAge()) * barWidth + 1;
506             int y1;
507             double v = info.getBalance(i);
508             int height;
509             int yOrigin = (int) (maxValue * scale);
510
511             if (v >= 0)
512             {
513                 y1 = (int) ((maxValue - v) * scale);
514                 height = yOrigin - y1;
515             }
516             else
517             {
518                 y1 = yOrigin;
519                 height = (int) (-v * scale);
520             }
521
522             if (i < info.getRetireAge()) g2.setPaint(colorPre);
523             else if (v >= 0) g2.setPaint(colorGain);
524             else g2.setPaint(colorLoss);
525             Rectangle2D bar = new Rectangle2D
526                             .Double(x1, y1, barWidth - 2, height);
527             g2.fill(bar);
528             g2.setPaint(Color.black);
529             g2.draw(bar);
530         }
531     }
532
533     /**
534      * Устанавливает цвет графика для периода до выхода на пенсию
535      * @param color Требующийся цвет
536      */
537     public void setColorPre(Color color)
538     {
539         colorPre = color;
```

```

540     repaint();
541 }
542
543 /**
544 * Устанавливает цвет графика для периода после
545 * выхода на пенсию, когда остаток на пенсионном
546 * счете еще положительный
547 * @param color Требующийся цвет
548 */
549 public void setColorGain(Color color)
550 {
551     colorGain = color;
552     repaint();
553 }
554
555 /**
556 * Устанавливает цвет графика для периода после
557 * выхода на пенсию, когда остаток на пенсионном
558 * счете уже отрицательный
559 * @param color Требующийся цвет
560 */
561 public void setColorLoss(Color color)
562 {
563     colorLoss = color;
564     repaint();
565 }
566
567 public Dimension getPreferredSize() { return PREFERRED_SIZE; }
568 }

```

Листинг 7.6. Исходный код из файла *retire/RetireResources.java*

```

1 package retire;
2
3 import java.awt.*;
4
5 /**
6 * Нестрочные ресурсы для пользовательского интерфейса на
7 * английском языке программы калькуляции пенсионных сбережений
8 * @version 1.21 2001-08-27
9 * @author Cay Horstmann
10 */
11 public class RetireResources extends java.util.ListResourceBundle
12 {
13     private static final Object[][] contents = {
14         // НАЧАЛО ИНТЕРНАЦИОНАЛИЗАЦИИ
15         { "colorPre", Color.blue }, { "colorGain", Color.white },
16         { "colorLoss", Color.red }
17     // КОНЕЦ ИНТЕРНАЦИОНАЛИЗАЦИИ
18     };
19
20     public Object[][] getContents()
21     {
22         return contents;
23     }
24 }

```

Листинг 7.7. Исходный код из файла retire/RetireResources_de.java

```
1 package retire;
2
3 import java.awt.*;
4
5 /**
6  * Нестрочные ресурсы для пользовательского интерфейса на
7  * немецком языке программы калькуляции пенсионных сбережений
8  * @version 1.21 2001-08-27
9  * @author Cay Horstmann
10 */
11 public class RetireResources_de extends
12     java.util.ListResourceBundle
13 {
14     private static final Object[][] contents = {
15         // НАЧАЛО ИНТЕРНАЦИОНАЛИЗАЦИИ
16         { "colorPre", Color.yellow }, { "colorGain", Color.black },
17         { "colorLoss", Color.red }
18     // КОНЕЦ ИНТЕРНАЦИОНАЛИЗАЦИИ
19     };
20
21     public Object[][] getContents()
22     {
23         return contents;
24     }
25 }
```

Листинг 7.8. Исходный код из файла retire/RetireResources_zh.java

```
1 package retire;
2
3 import java.awt.*;
4
5 /**
6  * Нестрочные ресурсы для пользовательского интерфейса на
7  * китайском языке программы калькуляции пенсионных сбережений
8  * @version 1.21 2001-08-27
9  * @author Cay Horstmann
10 */
11 public class RetireResources_zh extends
12     java.util.ListResourceBundle
13 {
14     private static final Object[][] contents = {
15         // НАЧАЛО ИНТЕРНАЦИОНАЛИЗАЦИИ
16         { "colorPre", Color.red }, { "colorGain", Color.blue },
17         { "colorLoss", Color.yellow }
18     // КОНЕЦ ИНТЕРНАЦИОНАЛИЗАЦИИ
19     };
20
21     public Object[][] getContents()
22     {
23         return contents;
24     }
25 }
```

Листинг 7.9. Содержимое файла свойств *retire/RetireStrings.properties*

```

1 language=Language
2 computeButton=Compute
3 savings=Prior Savings
4 contrib=Annual Contribution
5 income=Retirement Income
6 currentAge=Current Age
7 retireAge=Retirement Age
8 deathAge=Life Expectancy
9 inflationPercent=Inflation
10 investPercent=Investment Return
11 retire=Age: {0,number} Balance: {1,number,currency}

```

Листинг 7.10. Содержимое файла свойств *retire/RetireStrings_de.properties*

```

1 language=Sprache
2 computeButton=Rechnen
3 savings=Vorherige Ersparnisse
4 contrib=J\u00e4hrliche Einzahlung
5 income=Einkommen nach Ruhestand
6 currentAge=Jetziges Alter
7 retireAge=Ruhestandsalter
8 deathAge=Lebenserwartung
9 inflationPercent=Inflation
10 investPercent=Investitionsgewinn
11 retire=Alter: {0,number} Guthaben: {1,number,currency}

```

Листинг 7.11. Содержимое файла свойств *retire/RetireStrings_zh.properties*

```

1 language=\u8bed\u8a00
2 computeButton=\u8ba1\u7b97
3 savings=\u65e2\u5b58
4 contrib=\u6bcf\u5e74\u5b58\u91d1
5 income=\u9000\u4f11\u6536\u5165
6 currentAge=\u73b0\u9f84
7 retireAge=\u9000\u4f11\u5e74\u9f84
8 deathAge=\u9884\u671f\u5bff\u547d
9 inflationPercent=\u901a\u8d27\u81a8\u6da8
10 investPercent=\u6295\u8d44\u62a5\u916c
11 retire=\u5e74\u9f84: {0,number} \u603b\u7ed3: {1,number,currency}

```

Теперь вам должно быть понятно, как пользоваться средствами интернационализации в Java. В частности, для перевода текстовой информации на многие языки служат комплекты ресурсов, а средства форматирования и сортировки применяются для обработки текста с учетом региональных настроек. В следующей главе будут рассмотрены вопросы написания сценариев, компиляции и обработки аннотаций.

Написание сценариев, компиляция и обработка аннотаций

В этой главе...

- ▶ Написание сценариев для платформы Java
- ▶ Прикладной программный интерфейс API для компилятора
- ▶ Применение аннотаций
- ▶ Синтаксис аннотаций
- ▶ Стандартные аннотации
- ▶ Обработка аннотаций на уровне исходного кода
- ▶ Конструирование байт-кодов

В этой главе рассматриваются три методики обработки кода. Прикладной программный интерфейс API для создания сценариев позволяет вызывать код на языке сценариев таким же образом, как и в языке JavaScript или Groovy, прикладной программный интерфейс API для компилятора дает возможность компилировать исходный код Java в самой прикладной программе, а обработчики аннотаций — обрабатывать файлы классов или исходного кода Java, содержащие аннотации. Из этой главы вы узнаете, что существует немало приложений для обработки аннотаций, начиная с простой диагностики и заканчивая так называемым “конструированием байт-кодов”, вставкой байт-кодов в файлы классов и даже выполнением программ.

8.1. Написание сценариев для платформы Java

Язык сценариев — это такой язык программирования, который позволяет избегать обычного цикла операций редактирования, компиляции, компоновки и выполнения благодаря интерпретации исходного текста программы во время выполнения. Языки сценариев обладают рядом следующих преимуществ.

- Быстрый цикл обработки, стимулирующий стремление к экспериментированию.
- Возможность изменять поведение выполняющейся программы.
- Возможность для пользователей специально настраивать программы.

С другой стороны, у большинства языков сценариев отсутствуют средства, необходимые для разработки сложных прикладных программ, включая строгий контроль типов, инкапсуляцию и модульность.

В связи с этим возникает соблазн объединить преимущества языков сценариев с преимуществами традиционных языков программирования. Именно это и позволяет сделать прикладной программный интерфейс API для создания сценариев на платформе Java. В частности, он предоставляет возможность вызывать из программы на Java сценарии, написанные на JavaScript, Groovy, Ruby и даже таких экзотических языках, как Scheme и Haskell. Например, в проекте Renjin (www.renjin.org) предоставляется реализованная в Java возможность программировать на языке R, который зачастую применяется в области статистического программирования. С этой целью используется “механизм” прикладного программного интерфейса API для написания сценариев.

В последующих разделах будет показано, как выбирается механизм выполнения сценариев для конкретного языка, как выполняются сценарии и как пользоваться дополнительными преимуществами, которые дают некоторые механизмы сценариев.

8.1.1. Получение механизма сценариев

Механизм сценариев — это, по существу, библиотека, позволяющая выполнять сценарии на конкретном языке. При запуске виртуальная машина обнаруживает все доступные механизмы сценариев. Получить их перечень можно, создав объект типа `ScriptEngineManager` и вызвав метод `getEngineFactories()`. Далее у каждой фабрики механизмов можно запросить сведения об именах поддерживаемых механизмов, типах MIME и расширениях файлов. В табл. 8.1 перечислены наиболее употребительные механизмы сценариев и соответствующие им типы и расширения.

Обычно требующийся механизм известен и может запрашиваться просто по имени, типу MIME или расширению, как показано в приведенном ниже примере.

```
ScriptEngine engine = manager.getEngineByName("nashorn");
```

В версии Java SE 8 внедрен механизм Nashorn, разработанный в компании Oracle как вариант интерпретатора JavaScript. Предоставив необходимые архивные JAR-файлы по пути к соответствующим классам, можно дополнить перечень языков написания сценариев.

Таблица 8.1. Свойства фабрик механизмов сценариев

Механизм	Имена	Типы MIME	Расширения
Nashorn (входит в состав Java SE)	nashorn, Nashorn, js, JS, JavaScript, javascript, ECMAScript, ecmascript	application/javascript, application/ecmascript, text/javascript, text/ecmascript	.js
Groovy	groovy	Отсутствуют	groovy
Renjin	Renjin	text/x-R	R, r, S, s
SISC Scheme	sisc	Отсутствуют	scheme, sisc

javax.script.ScriptEngineManager 6

- **List<ScriptEngineFactory> getEngineFactories()**
Получает список всех обнаруженных фабрик механизмов сценариев.
- **ScriptEngine getEngineByName(String name)**
- **ScriptEngine getEngineByExtension(String extension)**
- **ScriptEngine getEngineByMimeType(String mimeType)**
Получают механизм сценариев с заданным именем, расширением файла сценария или типом MIME.

javax.script.ScriptEngineFactory 6

- **List<String> getNames()**
- **List<String> getExtensions()**
- **List<String> getMimeTypes()**
Получают имена, расширения файлов сценариев и типы MIME, по которым известна данная фабрика.

8.1.2. Выполнение сценариев и привязки

Получив механизм, можно приступить к вызову сценария. Это делается следующим образом:

```
Object result = engine.eval(scriptString);
```

Если сценарий хранится в файле, необходимо открыть поток чтения типа Reader и сделать следующий вызов:

```
Object result = engine.eval(reader);
```

С помощью одного и того же механизма можно вызвать целый ряд сценариев. Если какой-нибудь из сценариев содержит определения переменных, функций или классов, большинство механизмов сценариев будет сохранять их для последующего использования. Так, в приведенном ниже примере кода возвращается значение 1729.

```
engine.eval("n = 1728");
Object result = engine.eval("n + 1");
```



На заметку! Чтобы выяснить, является ли безопасным параллельное выполнение сценариев во многих потоках, достаточно сделать следующий вызов:

```
Object param = factory.getParameter("THREADING");
```

В итоге возвращается одно из перечисленных ниже значений.

- **null**: параллельное выполнение небезопасно.
- **"MULTITHREADED"**: параллельное выполнение безопасно. Результаты выполнения одного потока исполнения могут быть доступны из другого потока.
- **"THREAD-ISOLATED"**: то же, что и значение **"MULTITHREADED"**, но только для каждого потока исполнения поддерживаются разные привязки переменных.
- **"STATELESS"**: то же, что и значение **"THREAD-ISOLATED"**, но только сценарии не могут изменять привязки переменных.

Механизм сценариев нередко требуется дополнять привязками переменных. Каждая привязка состоит из имени и связываемого объекта Java. Рассмотрим в качестве примера следующие операторы:

```
engine.put(k, 1728);  
Object result = engine.eval("k + 1");
```

Код сценария читает определение объекта `k` из привязок в “области действия механизма”. И это очень важно, ведь почти все языки сценариев могут получать доступ к объектам Java и зачастую посредством более простого, чем у Java, синтаксиса. Например:

```
engine.put(b, new JButton());  
engine.eval("b.text = 'Ok'");
```

С другой стороны, можно извлекать значения переменных, привязанных операторами сценария, как показано ниже.

```
engine.eval("n = 1728");  
Object result = engine.get("n");
```

Кроме области действия механизма, существует и глобальная область действия. Любые привязки, которые вводятся в объект типа `ScriptEngineManager`, становятся видимыми для всех механизмов.

Вместо того чтобы вводить привязки в глобальную область действия или область действия механизма сценариев, их можно накапливать в объекте типа `Bindings` и передавать методу `eval()`, как показано ниже. Это очень удобно, если набор привязок не требуется сохранять для последующих вызовов метода `eval()`.

```
Bindings scope = engine.createBindings();  
scope.put(b, new JButton());  
engine.eval(scriptString, scope);
```



На заметку! Безусловно, может возникнуть потребность иметь и другие области действия, в отличие от глобальной области действия, а также области действия механизма сценариев. Например, веб-контейнеру могут потребоваться области действия запросов и сеансов. В подобных случаях разработчикам приходится самостоятельно создавать класс, реализующий интерфейс `ScriptContext`, чтобы управлять набором своих областей действия. Каждая такая область действия должна снабжаться целочисленным номером, а поиск должен выполняться в первую очередь в областях действия с наименьшим номером. [В стандартной библиотеке доступен только класс `SimpleScriptContext`, но он предусматривает лишь глобальную область действия, а также область действия механизма сценариев.]

javax.script.ScriptEngine 6

- **Object eval(String script)**
- **Object eval(Reader reader)**
- **Object eval(String script, Bindings bindings)**
- **Object eval(Reader reader, Bindings bindings)**

Оценивают сценарий, предоставляемый в символьной строке или средством чтения с учетом заданных привязок.

- **Object get(String key)**
- **void put(String key, Object value)**

Получают или размещают привязку в области действия механизма сценариев.

javax.script.ScriptEngineManager 6

- **Object get(String key)**
- **void put(String key, Object value)**

Получают или размещают привязку в глобальной области действия.

javax.script.Bindings 6

- **Object get(String key)**
- **void put(String key, Object value)**

Получают или размещают в области действия привязку, представляемую данным объектом типа **Bindings**.

8.1.3. Переадресация ввода-вывода

Стандартный ввод-вывод в сценарии можно переадресовывать, вызывая метод **setReader()** или **setWriter()** соответственно в контексте сценария, как показано в приведенном ниже примере кода, где любые данные, выводимые с помощью таких функций JavaScript, как **print()** или **println()**, направляются объекту **writer**.

```
StringWriter writer = new StringWriter();
engine.getContext().setWriter(new PrintWriter(writer, true));
```

Методы **setReader()** и **setWriter()** воздействуют только на стандартные источники ввода-вывода данных в механизме сценариев. Например, при выполнении приведенного ниже кода в сценарии JavaScript перенаправлен будет только первый вывод. Механизму сценариев Nashorn ничего неизвестно о стандартном источнике ввода данных, поэтому вызов метода **setReader()** ничего не даст.

```
println("Hello");
java.lang.System.out.println("World");
```

javax.script.ScriptEngine 6

- **ScriptContext getContext()**

Получает стандартный контекст сценариев для данного механизма.

javax.script.ScriptContext 6

- **Reader getReader()**
- **void setReader(Reader reader)**
- **Writer getWriter()**
- **void setWriter(Writer writer)**
- **Writer getErrorWriter()**
- **void setErrorWriter(Writer writer)**

Получают или устанавливают поток чтения для вводимых данных или поток записи для обычных или уведомляющих об ошибках выводимых данных.

8.1.4. Вызов сценарийных функций и методов

При наличии многих механизмов сценариев функция может вызываться на языке сценариев и без оценки фактического кода сценария. Это удобно, если пользователям разрешается реализовывать службу на избранном ими языке сценариев.

Те механизмы сценариев, которые предоставляют подобные функциональные возможности, реализуют интерфейс `Invocable`. В частности, этот интерфейс реализует механизм сценариев `Nashorn`. Чтобы вызвать сценарийную функцию, достаточно обратиться к методу `invokeFunction()`, указав в нем имя и параметры требуемой функции:

```
// определить функцию приветствия в JavaScript
engine.eval("function greet(how, whom)
    { return how + ', ' + whom + '!' }");
// вызвать эту функцию с аргументами "Hello", "World"
result = ((Invocable) engine).invokeFunction(
            "greet", "Hello", "World");
```

Если же язык сценариев является объектно-ориентированным, можно вызвать метод `invokeMethod()` следующим образом:

```
// определить класс Greeter в JavaScript
engine.eval("function Greeter(how) { this.how = how }");
engine.eval("Greeter.prototype.welcome =
    + " function(whom) { return this.how + ', ' + whom + '!' }");

// построить экземпляр
Object yo = engine.eval("new Greeter('Yo')");

// вызвать метод приветствия для экземпляра
result = ((Invocable) engine).invokeMethod(yo, "welcome", "World");
```



На заметку! Подробнее об определении классов в JavaScript см. в книге *JavaScript — The Good Parts* Дугласа Крокфорда [Douglas Crockford, издательство O'Reilly, 2008 г.; в русском переводе книга вышла под названием *JavaScript. Сильные стороны* в издательстве "Питер", 2012 г.]



На заметку! Даже если механизм сценариев не реализует интерфейс `Invocable`, метод все равно можно вызвать не зависящим от языка образом. Метод `getMethodCallSyntax()` из класса `ScriptEngineFactory` формирует символьную строку, которую можно затем передать методу `eval()`. Но все параметры этого метода должны быть привязаны к именам, в то время как метод `invokeMethod()` может вызываться с произвольными значениями параметров.

Можно пойти еще дальше и запросить механизм сценариев реализовать интерфейс Java. В этом случае появится возможность вызывать сценарные функции и методы, используя синтаксис Java для вызова методов. И хотя это зависит от конкретного механизма сценариев, как правило, для каждого метода из интерфейса достаточно предоставить соответствующую функцию. Рассмотрим в качестве примера следующий интерфейс Java:

```
public interface Greeter
{
    String greet(String whom);
}
```

Если определить глобальную функцию с тем же самым именем в Nashorn, ее можно вызвать через следующий интерфейс:

```
// определить функцию приветствия в JavaScript
engine.eval("function welcome(whom)
            { return 'Hello, ' + whom + '!' }");
// получить объект Java и вызвать метод Java
Greeter g = ((Invocable) engine).getInterface(Greeter.class);
result = g.welcome("World");
```

В объектно-ориентированном языке сценариев доступ к классу сценария можно получить через соответствующий интерфейс Java. В следующем примере кода демонстрируется, каким образом объект класса `SimpleGreeter` из языка JavaScript вызывается в синтаксисе языка Java:

```
Greeter g = ((Invocable) engine).getInterface(yo, Greeter.class);
result = g.welcome("World");
```

Таким образом, интерфейс `Invocable` оказывается удобным в том случае, если требуется вызывать код сценария из кода Java, не особенно разбираясь в синтаксисе языка написания сценариев.

`javax.script.Invocable` 6

- `Object invokeFunction(String name, Object... parameters)`
- `Object invokeMethod(Object implicitParameter, String name, Object... explicitParameters)`

Вызывают функцию или метод с указанным именем, передавая заданные параметры.

- `<T> T getInterface(Class<T> iface)`

Возвращает реализацию указанного интерфейса, методы которого реализуются с помощью функций из механизма сценариев.

- `<T> T getInterface(Object implicitParameter, Class<T> iface)`

Возвращает реализацию указанного интерфейса, методы которого реализуются с помощью методов заданного объекта.

8.1.5. Компиляция сценариев

Некоторые механизмы сценариев способны компилировать код сценария в промежуточную форму для более эффективного выполнения. Такие механизмы реализуют интерфейс `Compilable`. В следующем примере кода демонстрируется компилирование и оценивание кода, содержащегося в файле сценария:

```
Reader reader = new FileReader("myscript.js");
CompiledScript script = null;
if (engine implements Compilable)
    CompiledScript script = ((Compilable) engine).compile(reader);
```

После компиляции сценария можно переходить к его выполнению. В приведенном ниже фрагменте кода демонстрируется выполнение скомпилированного кода сценария, если компиляция прошла успешно, а иначе — исходного сценария, если окажется, что механизм сценариев не поддерживает компиляцию. Безусловно, компилировать сценарий нужно лишь в том случае, если его требуется выполнить повторно.

```
if (script != null)
    script.eval();
else
    engine.eval(reader);
```

`javax.script.Compilable` 6

- `CompiledScript compile(String script)`
- `CompiledScript compile(Reader reader)`

Компилируют сценарий, задаваемый символьной строкой или потоком чтения.

`javax.script.CompiledScript` 6

- `Object eval()`
- `Object eval(Bindings bindings)`

Оценивают данный сценарий.

8.1.6. Пример создания сценария для обработки событий в ГПИ

Чтобы продемонстрировать возможности прикладного программного интерфейса API для создания сценариев, рассмотрим пример программы, дающей пользователям возможность задавать обработчики событий на избранном языке сценариев.

Проанализируйте исходный код, приведенный в листинге 8.1. В этом коде средства создания сценариев вводятся в класс произвольного фрейма. По умолчанию это класс `ButtonFrame` из листинга 8.2, аналогичный по своим функциям программе обработки событий, демонстрировавшейся в первом томе настоящего издания, но со следующими различиями:

- у каждого компонента имеется свой собственный набор свойств name;
- отсутствуют обработчики событий.

Требующиеся обработчики событий определяются в файле свойств, а каждое свойство определяется в следующей форме:

```
имяКомпонента.имяСобытия = кодСценария
```

Так, если пользователь выбирает язык написания сценариев JavaScript, требующиеся обработчики событий предстаиваются в файле `js.properties` приведенным ниже образом. В сопутствующем коде имеются также файлы для Groovy, R и SISC Scheme.

```
yellowButton.action=panel.background = java.awt.Color.YELLOW  
blueButton.action=panel.background = java.awt.Color.BLUE  
redButton.action=panel.background = java.awt.Color.RED
```

Рассматриваемая здесь программа начинается с загрузки механизма сценариев на языке, указываемом в командной строке. Если же язык не указан, то по умолчанию выбирается JavaScript.

Далее выполняется обработка сценария `init.язык`, если таковой имеется. В целом такой подход кажется вполне приемлемым. К тому же интерпретатор Scheme нуждается в ряде громоздких операций инициализации, которые вряд ли стоит включать в каждый сценарий для обработки событий.

После этого осуществляется рекурсивный обход всех дочерних компонентов и ввод привязок (имя, объект) в область действия механизма сценариев. Далее выполняется чтение из файла свойств `язык.properties`. Для каждого свойства конструируется прокси-объект, замещающий обработчик событий, который, собственно, и заставляет выполнятся код сценария. Подробности реализации механизма замещения носят несколько технический характер, поэтому тем, кто желает разобраться в нем, рекомендуется еще раз прочитать раздел, посвященный прокси-объектам в главе 6 первого тома настоящего издания. Но самое главное, что каждый обработчик событий вызывает следующий метод:

```
engine.eval(scriptCode);
```

Остановимся подробнее на обработке событий от кнопки выбора желтого цвета фона (объекте `yellowButton`). При обработке приведенной ниже строки кода обнаруживается компонент JButton под именем "yellowButton".

```
yellowButton.action=panel.background = java.awt.Color.YELLOW
```

Далее к этому компоненту присоединяется объект типа `ActionListener` с методом `actionPerformed()`, который выполняет сценарий, если он создан средствами Nashorn:

```
panel.background = java.awt.Color.YELLOW
```

Механизм сценариев содержит привязку, которая связывает имя "panel" с объектом типа JPanel. Когда наступает событие, выполняется метод `setBackground()` для этого объекта, а в итоге изменяется цвет фона панели.

Запустить рассматриваемую здесь программу с обработчиками событий из сценария JavaScript можно, выполнив команду

```
java ScriptTest
```

А для того чтобы использовать обработчики событий из сценария Groovy, придется выполнить такую команду:

```
java -classpath .:groovy/lib/* ScriptTest groovy
```

где **groovy** — каталог, в котором установлен Groovy. А для реализации языка R по проекту Renjin архивные JAR-файлы для библиотеки Renjin Studio и механизма сценариев Renjin следует включить в путь к соответствующим классам. Оба эти компонента свободно доступны для загрузки по адресу www.renjin.org/downloads.html.

И наконец, чтобы опробовать Scheme, следует загрузить SISC Scheme с веб-сайта по адресу <http://sisc-scheme.org/> и выполнить следующую команду:

```
java -classpath .:sisc/*:jsr223-engines/scheme/build/scheme-engine.jar  
      ScriptTest scheme
```

где **sisc** — каталог установки SISC Scheme, а **jsr223-engines** — каталог, содержащий адаптеры механизма сценариев, доступные по адресу <http://java.net/projects/scripting>.

В рассматриваемом здесь примере программы демонстрируется применение сценариев при программировании ГПИ на платформе Java. Желающие могут пойти еще дальше и описать ГПИ с помощью XML-файла, как было показано в главе 3. В этом случае данная программа превратится в интерпретатор для ГПИ с визуальным представлением, определяемым в формате XML, а также поведением, определяемым на языке сценариев. Это очень похоже на среду создания динамических серверных сценариев и динамических HTML-страниц.

Листинг 8.1. Исходный код из файла script/ScriptTest.java

```
1  package script;  
2  
3  import java.awt.*;  
4  import java.beans.*;  
5  import java.io.*;  
6  import java.lang.reflect.*;  
7  import java.util.*;  
8  import javax.script.*;  
9  import javax.swing.*;  
10  
11  /**  
12   * @version 1.02 2016-05-10  
13   * @author Cay Horstmann  
14  */  
15  public class ScriptTest  
16  {  
17      public static void main(String[] args)  
18      {  
19          EventQueue.invokeLater(() ->  
20          {  
21              try  
22              {  
23                  ScriptEngineManager manager = new ScriptEngineManager();  
24                  String language;  
25                  if (args.length == 0)  
26                  {  
27                      System.out.println("Available factories: ");  
28                      for (ScriptEngineFactory factory :  
29                          manager.getEngineFactories())  
30                          System.out.println(factory.getEngineName());  
31              }  
32          }  
33      }  
34  }
```

```
31             language = "nashorn";
32     }
33     else language = args[0];
34
35     final ScriptEngine engine =
36         manager.getEngineByName(language);
37     if (engine == null)
38     {
39         System.err.println("No engine for " + language);
40         System.exit(1);
41     }
42
43     final String frameClassName = args.length < 2
44         ? "buttons1.ButtonFrame" : args[1];
45     JFrame frame = (JFrame)
46         Class.forName(frameClassName).newInstance();
47     InputStream in = frame.getClass().getResourceAsStream(
48         "init." + language);
49     if (in != null) engine.eval(new InputStreamReader(in));
50     Map<String, Component> components = new HashMap<>();
51     getComponentBindings(frame, components);
52     components.forEach((name, c) -> engine.put(name, c));
53
54     final Properties events = new Properties();
55     in = frame.getClass().getResourceAsStream(language
56                                         + ".properties");
57     events.load(in);
58
59     for (final Object e : events.keySet())
60     {
61         String[] s = ((String) e).split("\\.");
62         addListener(s[0], s[1], (String) events.get(e),
63                         engine, components);
64     }
65     frame.setTitle("ScriptTest");
66     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
67     frame.setVisible(true);
68 }
69 catch (ReflectiveOperationException | IOException
70         | ScriptException | IntrospectionException ex)
71 {
72     ex.printStackTrace();
73 }
74 });
75 }
76 }
77 */
78 /**
79 * Собирает все именованные компоненты в контейнер
80 * @param c Компонент
81 * @param namedComponents Отображение, в которое вводятся
82 *                       компоненты и их имена
83 */
84 private static void getComponentBindings(Component c,
85                                         Map<String, Component> namedComponents)
86 {
87     String name = c.getName();
88     if (name != null) { namedComponents.put(name, c); }
```

```

89     if (c instanceof Container)
90     {
91         for (Component child : ((Container) c).getComponents())
92             getComponentBindings(child, namedComponents);
93     }
94 }
95
96 /**
97 * Вводит в объект приемник событий, метод которого
98 * выполняет сценарий
99 * @param beanName Имя компонента JavaBeans, в который вводится
100 *                  приемник событий
101 * @param eventName Название типа приемника событий, например,
102 *                  "action" (действие) или "change" (изменение)
103 * @param scriptCode Выполняемый код сценария
104 * @param engine Механизм сценариев, выполняющий код сценария
105 * @param bindings Привязки для выполнения сценария
106 * @throws Генерирует исключение типа IntrospectionException
107 */
108 private static void addListener(String beanName,
109                                 String eventName, final String scriptCode,
110                                 final ScriptEngine engine,
111                                 Map<String,Component> components)
112                         throws ReflectiveOperationException, IntrospectionException
113 {
114     Object bean = components.get(beanName);
115     EventSetDescriptor descriptor = getEventSetDescriptor(
116                                         bean, eventName);
117     if (descriptor == null) return;
118     descriptor.getAddListenerMethod().invoke(bean,
119                                                 Proxy.newProxyInstance(null, new Class[]
120                                                 { descriptor.getListenerType() },
121                                                 (proxy, method, args) ->
122                                                 {
123                                                     engine.eval(scriptCode);
124                                                     return null;
125                                                 }
126                                         ));
127 }
128
129 private static EventSetDescriptor getEventSetDescriptor(
130             Object bean, String eventName)
131             throws IntrospectionException
132 {
133     for (EventSetDescriptor descriptor :
134             Introspector.getBeanInfo(bean.getClass())
135             .getEventSetDescriptors())
136         if (descriptor.getName().equals(eventName))
137             return descriptor;
138     return null;
139 }
140 }
```

Листинг 8.2. Исходный код из файла buttons1/ButtonFrame.java

```

1 package buttons1;
2
```

```
3 import javax.swing.*;
4
5 /**
6  * Фрейм с панелью кнопок
7  * @version 1.00 2007-11-02
8  * @author Cay Horstmann
9 */
10 public class ButtonFrame extends JFrame
11 {
12     private static final int DEFAULT_WIDTH = 300;
13     private static final int DEFAULT_HEIGHT = 200;
14
15     private JPanel panel;
16     private JButton yellowButton;
17     private JButton blueButton;
18     private JButton redButton;
19
20     public ButtonFrame()
21     {
22         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
23
24         panel = new JPanel();
25         panel.setName("panel");
26         add(panel);
27
28         yellowButton = new JButton("Yellow");
29         yellowButton.setName("yellowButton");
30         blueButton = new JButton("Blue");
31         blueButton.setName("blueButton");
32         redButton = new JButton("Red");
33         redButton.setName("redButton");
34
35         panel.add(yellowButton);
36         panel.add(blueButton);
37         panel.add(redButton);
38     }
39 }
```

8.2. Прикладной программный интерфейс API для компилятора

В предыдущих разделах было показано, как взаимодействовать с кодом на языке написания сценариев. А теперь рассмотрим другой вариант: программы на Java, компилирующие код Java. Имеется немало инструментальных средств, в которых требуется вызывать компилятор Java. К их числу относятся следующие.

- Среды разработки.
- Программы, обучающие программированию на Java.
- Автоматизированные средства разработки и тестирования.
- Инструментальные средства создания по шаблону, обрабатывающие фрагменты кода Java, например страницы JSP (JavaServer Pages).

Раньше приложения вызывали компилятор Java, обращаясь к недокументированным классам из библиотеки *jdk/lib/tools.jar*. В настоящее время общедоступный прикладной программный интерфейс API для компиляции является

частью платформы Java, и поэтому пользоваться библиотекой tools.jar больше не требуется. Об этом прикладном программном интерфейсе и пойдет речь в данном разделе.

8.2.1. Простой способ компилирования

Компилятор вызывается очень просто, как показано в приведенном ниже примере кода. Получаемое в итоге нулевое значение указывает на то, что компиляция прошла успешно.

```
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
OutputStream outStream = ....;
OutputStream errStream = ....;
int result = compiler.run(null, outStream, errStream, "-sourcepath",
    "src", "Test.java");
```

Все выводимые данные и сообщения об ошибках компилятор направляет в указанные потоки вывода. В качестве параметров метода run() можно указывать и пустое значение null. В данном случае используются стандартные потоки вывода System.out и System.err. Первый параметр метода run() обозначает поток ввода, но, поскольку никаких данных, вводимых с консоли, компилятор не принимает, значение этого параметра всегда оставляется пустым (null). Сам же метод run() наследуется из обобщенного интерфейса Tool, допускающего применение инструментальных средств для чтения вводимых данных. А остальные параметры метода run() являются просто аргументами, которые следовало было бы передать утилите javac, если бы этот метод вызывался из командной строки. Они могут обозначать как параметры командной строки, так и имена файлов.

8.2.2. Выполнение заданий на компиляцию

С помощью объекта типа CompilationTask можно получить еще больший контроль над процессом компиляции. Он, в частности, позволяет выполнять следующие действия.

- Управлять источником программного кода, предоставляя, например, код не в файле, а в построителе символьных строк.
- Управлять размещением файлов классов, сохраняя их, например, в базе данных.
- Принимать предупреждения и сообщения об ошибках, появляющиеся во время компиляции.
- Запускать компилятор на выполнение в фоновом режиме.

За размещение исходных файлов и файлов классов отвечает интерфейс JavaFileManager. В его обязанности входит определение для этих файлов экземпляров типа JavaFileObject. Каждый объект типа JavaFileObject может соответствовать какому-нибудь определенному файлу на диске или предоставлять какой-нибудь другой механизм для чтения и записи содержимого файла.

Для приема появляющихся сообщений об ошибках устанавливается интерфейс DiagnosticListener. Всякий раз, когда компилятор выдает предупреждение или сообщение об ошибке, этот приемник получает объект типа Diagnostic. Этот

интерфейс реализуется классом `DiagnosticCollector`. Он просто собирает все диагностические данные для просмотра и анализа по завершении компиляции.

Объект типа `Diagnostic` содержит сведения о месте появления ошибки компиляции, включая имя файла, номер строки и столбца, а также удобочитаемое описание ошибки. А для получения объекта типа `CompilationTask` вызывается метод `getTask()` из класса `JavaCompiler` с указанием следующих аргументов.

- Объект типа `Writer` для вывода любых данных компилятора, не являющихся диагностическими (типа `Diagnostic`), или пустого значения `null` в стандартный поток вывода ошибок `System.err`.
- Объект типа `JavaFileManager` или пустое значение `null` для использования стандартного диспетчера файлов компилятора.
- Объект типа `DiagnosticListener`.
- Символьная строка параметров командной строки или пустое значение `null`, если никаких параметров указывать не требуется.
- Имена классов для обработки аннотаций или пустые значения `null`, если аннотации отсутствуют (подробнее обработка аннотаций рассматривается далее в этой главе).
- Экземпляры типа `JavaFileObject` для исходных файлов.

Последние три аргумента необходимо предоставить в виде объектов типа `Iterable`. Например, последовательность параметров командной строки можно указать следующим образом:

```
Iterable<String> options = Arrays.asList("-g", "-d", "classes");
```

С другой стороны, можно воспользоваться любым из классов коллекций. Если же требуется, чтобы компилятор считывал исходные файлы с диска, то можно обратиться к интерфейсу `StandardJavaFileManager`, чтобы преобразовать символьные строки с именами файлов или объекты типа `File` в экземпляры типа `JavaFileObject`, как показано в приведенном ниже примере.

```
StandardJavaFileManager fileManager =
    compiler.getStandardFileManager(null, null, null);
Iterable<JavaFileObject> fileObjects =
    fileManager.getJavaFileObjectsFromStrings(fileName);
```

Но если требуется, чтобы компилятор считывал исходный код из какого-нибудь другого места, а не из файла на диске, то придется предоставить свой собственный подкласс, реализующий интерфейс `JavaFileObject`. В листинге 8.3 приведен исходный код класса для получения объекта исходного файла с данными, содержащимися в объекте типа `StringBuilder`. Этот класс расширяет простой класс `SimpleJavaFileObject` и переопределяет метод `getCharContent()`, чтобы возвращать содержимое построителя символьных строк. Он применяется в рассматриваемом здесь примере программы, где код класса `Java` сначала генерируется динамически, а затем компилируется.

Класс `CompilationTask` реализует интерфейс `Callable<Boolean>`. Его можно передать классу `Executor` для выполнения в другом потоке или просто вызвать с помощью метода `call()`. Возвращаемое значение `Boolean.FALSE` обозначает

неудачный исход компиляции. Применение данного класса демонстрируется в приведенном ниже фрагменте кода.

```
Callable<Boolean> task = new JavaCompiler.CompilationTask(null,
    fileManager, diagnostics, options, null, fileObjects);
if (!task.call())
    System.out.println("Compilation failed");
```

Если только требуется, чтобы компилятор создавал файлы классов и сохранял их на диске, то выполнять специальную настройку интерфейса JavaFileManager не нужно. Тем не менее в рассматриваемом здесь примере программы файлы классов формируются сначала в виде байтовых массивов, а затемчитываются из памяти с помощью специального загрузчика классов. В листинге 8.4 приведен исходный код класса, реализующего интерфейс JavaFileObject. Его метод openOutputStream() возвращает поток вывода типа ByteArrayOutputStream, в который компилятор направляет байт-коды.

Оказывается, что предписать диспетчеру файлов компилятора использовать упомянутые выше объекты файлов не так-то просто. Дело в том, что в стандартной библиотеке отсутствует класс, который реализовывал бы интерфейс StandardJavaFileManager. Поэтому приходится создавать подкласс, производный от класса ForwardingJavaFileManager и делегирующий все вызовы заданному диспетчеру файлов. В данном примере требуется лишь изменить метод getJavaFileForOutput(), и достичь этой цели можно с помощью следующего кода:

```
JavaFileManager fileManager = compiler
    .getStandardFileManager(diagnostics, null, null);
fileManager =
    new ForwardingJavaFileManager<JavaFileManager>(fileManager)
{
    public JavaFileObject getJavaFileForOutput(
        Location location, final String className, Kind kind,
        FileObject sibling) throws IOException
    {
        // возвратить специальный объект файла
    }
};
```

Таким образом, вызывать метод run() для задания на компиляцию типа JavaCompiler удобно только в тех случаях, когда требуется активизировать компилятор обычным образом и выполнять операции чтения и записи файлов на диск. Выводимые данные и сообщения об ошибках можно перехватывать, но выполнять их синтаксический анализ придется самостоятельно.

Если же требуется больший контроль над процессами обработки файлов и уведомления об ошибках, то лучше воспользоваться классом CompilationTask. Его прикладной программный интерфейс API довольно сложен, но в то же время он позволяет управлять каждым аспектом процесса компиляции.

Листинг 8.3. Исходный код из файла compiler/StringBuilderJavaSource.java

```
1 package compiler;
2
3 import java.net.*;
```

```
4 import javax.tools.*;
5
6 /**
7  * Источник, хранящий код Java в построителе символьных строк
8  * @version 1.00 2007-11-02
9  * @author Cay Horstmann
10 */
11 public class StringBuilderJavaSource extends SimpleJavaFileObject
12 {
13     private StringBuilder code;
14
15     /**
16      * Конструирует новый объект типа StringBuilderJavaSource
17      * @param name Имя исходного файла, представленного
18      *             данным объектом
19      */
20     public StringBuilderJavaSource(String name)
21     {
22         super(URI.create("string:///+" + name.replace('.', '/') +
23                     + Kind.SOURCE.extension), Kind.SOURCE);
24         code = new StringBuilder();
25     }
26
27     public CharSequence getCharContent(boolean ignoreEncodingErrors)
28     {
29         return code;
30     }
31
32     public void append(String str)
33     {
34         code.append(str);
35         code.append('\n');
36     }
37 }
```

Листинг 8.4. Исходный код из файла compiler/ByteArrayJavaClass.java

```
1 package compiler;
2
3 import java.io.*;
4 import java.net.*;
5 import javax.tools.*;
6 /**
7  * Класс Java для хранения байт-кодов в байтовом массиве
8  * @version 1.00 2007-11-02
9  * @author Cay Horstmann
10 */
11 public class ByteArrayJavaClass extends SimpleJavaFileObject
12 {
13     private ByteArrayOutputStream stream;
14
15     /**
16      * Конструирует новый объект типа ByteArrayJavaClass
17      * @param name Имя файла класса, представленного данным объектом
18      */
19     public ByteArrayJavaClass(String name)
20     {
```

```

21     super(URI.create("bytes:/// " + name), Kind.CLASS);
22     stream = new ByteArrayOutputStream();
23 }
24
25 public OutputStream openOutputStream() throws IOException
26 {
27     return stream;
28 }
29
30 public byte[] getBytes()
31 {
32     return stream.toByteArray();
33 }
34 }
```

javax.tools.Tool 6

- int run(InputStream in, OutputStream out,
OutputStream err, String... arguments)**

Запускает утилиту компиляции с указанными потоками ввода и вывода, а также потоком вывода сообщений об ошибках и аргументами командной строки. При удачном исходе компиляции возвращает нулевое значение, а при неудачном исходе — ненулевое.

javax.tools.JavaCompiler 6

- StandardJavaFileManager getStandardFileManager(DiagnosticListener<? super JavaFileObject> diagnosticListener, Locale locale, Charset charset)**

Получает стандартный диспетчер файлов для данного компилятора. Имеется возможность использовать стандартные сообщения об ошибках, региональные настройки и набор символов, указав в качестве соответствующих параметров пустые значения null.

- JavaCompiler.CompilationTask getTask(Writer out, JavaFileManager fileManager, DiagnosticListener<? super JavaFileObject> diagnosticListener, Iterable<String> options, Iterable<String> classesForAnnotationProcessing, Iterable<? extends JavaFileObject> sourceFiles)**

Получает задание на компиляцию, при вызове которого будут компилироваться указанные исходные файлы. Подробнее об этом см. в предыдущем разделе.

javax.tools.StandardJavaFileManager 6

- Iterable<? extends JavaFileObject> getJavaFileObjectsFromStrings(Iterable<String> fileNames)**
- Iterable<? extends JavaFileObject> getJavaFileObjectsFromFiles(Iterable<? extends File> files)**

Преобразуют последовательность имен файлов или файлов в последовательность экземпляров типа `JavaFileObject`.

javax.tools.JavaCompiler.CompilationTask 6

- **Boolean call()**

Выполняет задание на компиляцию.

javax.tools.DiagnosticCollector<S> 6

- **DiagnosticCollector<S> DiagnosticCollector()**
Создает пустой сборщик данных.
- **List<Diagnostic<? extends S>> getDiagnostics()**
Получает собранные диагностические данные.

javax.tools.Diagnostic<S> 6

- **S getSource()**
Получает исходный объект, связанный с данной процедурой диагностики.
- **Diagnostic.Kind getKind()**
Получает тип данной процедуры диагностики, принимающий значение одной из следующих констант: **ERROR**, **WARNING**, **MANDATORY_WARNING**, **NOTE** или **OTHER**.
- **String getMessage(Locale locale)**
Получает сообщение, описывающее ошибку, обнаруженную в данной процедуре диагностики. Имеется возможность использовать стандартные региональные настройки, передав пустое значение **null** в качестве соответствующего параметра.
- **long getLineNumber()**
- **long getColumnNumber()**
Получают местоположение ошибки, обнаруженной в данной процедуре диагностики.

javax.tools.SimpleJavaFileObject 6

- **CharSequence getCharContent(boolean ignoreEncodingErrors)**
Этот метод переопределяется для объекта файла, представляющего исходный файл и генерирующего исходный код.
- **OutputStream openOutputStream()**
Этот метод переопределяется для объекта файла, представляющего файл класса и формирующего поток вывода, в который можно направлять генерируемые байт-коды.

javax.tools.ForwardingJavaFileManager<M extends JavaFileManager> 6

- **protected ForwardingJavaFileManager(M fileManager)**
Создает объект типа **JavaFileManager**, делегирующий все вызовы указанному диспетчеру файлов.

```
javax.tools.ForwardingJavaFileManager
<M extends JavaFileManager> 6 (окончание)
```

- `FileObject getFileForOutput(JavaFileManager.Location location, String className, JavaFileObject.Kind kind, FileObject sibling)`
Вызов этого метода перехватывается, если требуется заменить объект файла для записи файлов классов. Параметр `kind` может принимать значение одной из следующих констант: `SOURCE`, `CLASS`, `HTML` или `OTHER`.

8.2.3. Пример динамического генерирования кода Java

В технологии JSP для динамических веб-страниц допускается сочетание разметки в коде HTML с фрагментами кода Java, как показано в следующем примере:

```
<p>The current date and time is <b><%=
    new java.util.Date() %></b>.</p>
```

Механизм JSP динамически компилирует код Java в сервлет. В рассматриваемом здесь примере прикладной программы демонстрируется более простой пример динамического генерирования кода Swing. Основной замысел состоит в том, чтобы воспользоваться построителем ГПИ с целью расположить компоненты во фрейме и обозначить их поведение во внешнем файле. В листинге 8.5 приведен очень простой пример класса фрейма, а в листинге 8.6 — код для обозначения действий кнопок. Следует заметить, что конструктор класса фрейма вызывает абстрактный метод `addEventHandlers()`. А генератор кода создает подкласс, реализующий метод `addEventHandlers()`, вводя приемник для обработки действий в каждой строке из файла свойств `action.properties`. (Распространение возможностей генерирования кода на другие виды событий оставляется читателю в качестве упражнения для самостоятельной проработки.)

Этот подкласс размещается в пакете под именем `x`, которое нигде больше не должно использоваться в данной программе. Сгенерированный код принимает следующую форму:

```
package x;
public class Frame extends ИмяСуперкласса
{
    protected void addEventHandlers()
    {
        ИмяКомпонента1.addActionListener(new
            java.awt.event.ActionListener()
        {
            public void actionPerformed(java.awt.event.ActionEvent)
            {
                код первого обработчика событий
            }
        });
        // повторить для остальных обработчиков событий ...
    }
}
```

Метод `buildSource()` в программе из листинга 8.7 служит для генерирования этого кода и его размещения в объекте типа `StringBuilderJavaSource`. А затем этот объект передается компилятору Java.

Создание объектов типа `ByteArrayJavaClass` для каждого класса из пакета `x` осуществляется методом `getJavaFileForOutput()` из класса `ForwardingJavaFileManager`. Эти объекты перехватывают файлы классов, формируемые при компиляции класса `x.Frame`. Метод `getJavaFileForOutput()` вводит каждый объект файла в список прежде, чем возвратить его, чтобы в дальнейшем можно было обнаружить байт-коды. Следует иметь в виду, что компиляция класса `x.Frame` приводит к генерированию файла для главного класса, а также отдельных файлов для каждого класса приемника событий.

После компиляции составляется отображение, в котором имена классов сопоставляются с массивами байт-кодов. Классы, хранящиеся в этом отображении, загружаются простым загрузчиком классов из листинга 8.8. Сначала происходит обращение к этому загрузчику для загрузки только что скомпилированного класса, а затем создается и отображается класс фрейма прикладной программы:

```
ClassLoader loader = new MapClassLoader(byteCodeMap);
Class<?> cl = loader.loadClass("x.Frame");
Frame frame = (JFrame) cl.newInstance();
frame.setVisible(true);
```

Если щелкнуть на кнопках, цвет фона изменится обычным образом. А для того чтобы убедиться, что действия кнопок действительно компилируются динамически, измените какую-нибудь из строк в файле свойств `action.properties`, например, следующим образом:

```
yellowButton=panel.setBackground(java.awt.Color.YELLOW);
yellowButton.setEnabled(false);
```

После этого запустите еще раз данную программу на выполнение. Если теперь щелкнуть на кнопке `Yellow`, она станет недоступной. Загляните также в каталоги с исходным кодом. Вы не обнаружите там ни исходных файлов, ни файлов классов из пакета `x`. Данный пример наглядно демонстрирует применение динамической компиляции вместе с сохранением файлов исходного кода и классов в оперативной памяти.

Листинг 8.5. Исходный код из файла `buttons2/ButtonFrame.java`

```
1 package buttons2;
2 import javax.swing.*;
3
4 /**
5  * Фрейм с панелью кнопок
6  * @version 1.00 2007-11-02
7  * @author Cay Horstmann
8 */
9 public abstract class ButtonFrame extends JFrame
10 {
11     public static final int DEFAULT_WIDTH = 300;
12     public static final int DEFAULT_HEIGHT = 200;
13
14     protected JPanel panel;
15     protected JButton yellowButton;
16     protected JButton blueButton;
17     protected JButton redButton;
18 }
```

```
19 protected abstract void addEventHandlers();
20
21 public ButtonFrame()
22 {
23     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
24
25     panel = new JPanel();
26     add(panel);
27
28     yellowButton = new JButton("Yellow");
29     blueButton = new JButton("Blue");
30     redButton = new JButton("Red");
31
32     panel.add(yellowButton);
33     panel.add(blueButton);
34     panel.add(redButton);
35
36     addEventHandlers();
37 }
38 }
```

Листинг 8.6. Исходный код из файла buttons2/action.properties

```
1 yellowButton=panel.setBackground(java.awt.Color.YELLOW);
2 blueButton=panel.setBackground(java.awt.Color.BLUE);
```

Листинг 8.7. Исходный код из файла compiler/CompilerTest.java

```
1 package compiler;
2
3 import java.awt.*;
4 import java.io.*;
5 import java.util.*;
6 import java.util.List;
7 import javax.swing.*;
8 import javax.tools.*;
9 import javax.tools.JavaFileObject.*;
10
11 /**
12  * @version 1.01 2016-05-10
13  * @author Cay Horstmann
14  */
15 public class CompilerTest
16 {
17     public static void main(final String[] args)
18             throws IOException, ClassNotFoundException
19     {
20         JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
21
22         final List<ByteArrayJavaClass> classFileObjects =
23             new ArrayList<>();
24
25         DiagnosticCollector<JavaFileObject> diagnostics =
26             new DiagnosticCollector<>();
27
28         JavaFileManager fileManager = compiler
29             .getStandardFileManager(diagnostics, null, null);
```

```
30     fileManager = new
31         ForwardingJavaFileManager<JavaFileManager>(fileManager)
32     {
33         public JavaFileObject getJavaFileForOutput(
34             Location location, final String className,
35             Kind kind, FileObject sibling) throws IOException
36     {
37         if (className.startsWith("x."))
38     {
39             ByteArrayJavaClass fileObject =
40                 new ByteArrayJavaClass(className);
41             classFileObjects.add(fileObject);
42             return fileObject;
43         }
44         else return super.getJavaFileForOutput(location,
45             className, kind, sibling);
46     }
47 };
48
49     String frameClassName = args.length == 0
50         ? "buttons2.ButtonFrame" : args[0];
51     JavaFileObject source = buildSource(frameClassName);
52     JavaCompiler.CompilationTask task = compiler.getTask(
53         null, fileManager, diagnostics, null,
54         null, Arrays.asList(source));
55     Boolean result = task.call();
56
57     for (Diagnostic<? extends JavaFileObject> d :
58         diagnostics.getDiagnostics())
59     {
60         System.out.println(d.getKind()
61             + ": " + d.getMessage(null));
62     }
63     fileManager.close();
64     if (!result)
65     {
66         System.out.println("Compilation failed.");
67         System.exit(1);
68     }
69
70     EventQueue.invokeLater(() ->
71     {
72         try
73     {
74         Map<String, byte[]> byteCodeMap = new HashMap<>();
75         for (ByteArrayJavaClass cl : classFileObjects)
76             byteCodeMap.put(cl.getName().substring(1),
77                             cl.getBytes());
78         ClassLoader loader = new MapClassLoader(byteCodeMap);
79         JFrame frame = (JFrame) loader.loadClass("x.Frame")
80             .newInstance();
81         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
82         frame.setTitle("CompilerTest");
83         frame.setVisible(true);
84     }
85     catch (Exception ex)
86     {
87         ex.printStackTrace();
88     }
89 });

}
```

```

90  /*
91   * Генерирует исходный код подкласса, реализующего
92   * метод addEventHandlers()
93   * @return Возвращает объект файла, содержащий исходный
94   *         код в построителе символьных строк
95   */
96  static JavaFileObject buildSource(String superClassSimpleName)
97      throws IOException, ClassNotFoundException
98  {
99      StringBuilderJavaSource source =
100         new StringBuilderJavaSource("x.Frame");
101     source.append("package x;\n");
102     source.append("public class Frame extends ");
103     source.append(superClassSimpleName + " {\n");
104     source.append("protected void addEventHandlers() {}");
105     final Properties props = new Properties();
106     props.load(Class.forName(superClassSimpleName)
107         .getResourceAsStream("action.properties"));
108     for (Map.Entry<Object, Object> e : props.entrySet())
109     {
110         String beanName = (String) e.getKey();
111         String eventCode = (String) e.getValue();
112         source.append(beanName + ".addActionListener(event -> {\n");
113         source.append(eventCode);
114         source.append("} );\n");
115     }
116     source.append("} }");
117     return source;
118 }
119 }
```

Листинг 8.8. Исходный код из файла compiler/MapClassLoader.java

```

1 package compiler;
2
3 import java.util.*;
4
5 /**
6  * Загрузчик классов, загружающий классы из отображения,
7  * где в качестве ключей служат имена классов, а
8  * соответствующих им значений – массивы байт-кодов
9  * @version 1.00 2007-11-02
10 * @author Cay Horstmann
11 */
12 public class MapClassLoader extends ClassLoader
13 {
14     private Map<String, byte[]> classes;
15
16     public MapClassLoader(Map<String, byte[]> classes)
17     {
18         this.classes = classes;
19     }
20
21     protected Class<?> findClass(String name)
22         throws ClassNotFoundException
23     {
24         byte[] classBytes = classes.get(name);
25         if (classBytes == null)
```

```
26     throw new ClassNotFoundException(name);
27     Class<?> cl = defineClass(name, classBytes, 0,
28                               classBytes.length);
29     if (cl == null) throw new ClassNotFoundException(name);
30     return cl;
31   }
32 }
```

8.3. Применение аннотаций

Аннотациями называются дескрипторы, которые разработчики вставляют в свой исходный код, чтобы их можно было обработать соответствующими инструментальными средствами. Эти инструментальные средства могут действовать как на уровне исходного кода, так и на уровне файлов классов, в которых компилятор размещает аннотации. Аннотации не влияют на способ компиляции программ. Компилятор Java генерирует одинаковые инструкции виртуальной машины как с аннотациями, так и без них.

Чтобы извлечь наибольшую пользу из аннотаций, необходимо выбрать подходящее средство обработки. В исходный код следует вводить такие аннотации, которые распознаются избранным средством обработки, способным правильно интерпретировать их и выполнять соответствующие действия над исходных кодом. У аннотаций существует немало областей применения, и поэтому их универсальность может поначалу вызывать недоразумения. Ниже перечислены некоторые из областей применения аннотаций.

- Автоматическое генерирование вспомогательных файлов, например, файлов дескрипторов развертывания или классов информации о компонентах JavaBeans.
- Автоматическое генерирование кода для тестирования, протоколирования, семантической обработки транзакций и т.д.

8.3.1. Введение в аннотации

Итак, начнем обсуждение аннотаций с основных понятий и продемонстрируем их практическое применение на конкретном примере, пометив методы как приемники событий для компонентов AWT и представив обработчик аннотаций, способный анализировать аннотации и подключать приемники событий. Далее мы подробно рассмотрим синтаксические правила. И, наконец, продемонстрируем два расширенных примера обработки аннотаций: первый — на уровне исходного кода, второй — на уровне файлов классов благодаря применению библиотеки Apache Bytecode Engineering Library и вставке в аннотированные методы дополнительных байт-кодов.

Ниже приведен пример объявления простой аннотации. В частности, аннотация `@Test` помечает метод `checkRandomInsertions()`.

```
public class MyClass
{
    ...
    @Test public void checkRandomInsertions()
```

Аннотация применяется в Java подобно модификатору и размещается перед аннотируемым элементом без точки с запятой. (*Модификатор* — это ключевое слово `вроде public или static.`) Перед именем каждой аннотации ставится знак `@` подобно тому, как это делается в комментариях к документам, формируемым средствами Javadoc. Но комментарии в Javadoc размещаются между разделителями `/**...*/`, тогда как аннотации являются частью исходного кода.

Сама аннотация `@Test` ничего не делает. Чтобы она могла приносить какую-то пользу, ей необходимо подходящее инструментальное средство. Например, инструментальное средство модульного тестирования JUnit 4 (доступное по адресу <http://junit.org>) способно вызвать все помеченные аннотацией `@Test` методы при тестировании класса. Другое инструментальное средство может удалять все тестовые методы из файла класса, чтобы исключить их из исходного кода программы после ее тестирования. Аннотации могут быть определены вместе со своими элементами, как показано в приведенном ниже примере кода.

```
@Test(timeout="10000")
```

Эти элементы могут обрабатываться инструментальными средствами, читающими аннотации. Элементы могут выглядеть и по-другому. Подробнее о них речь пойдет далее в этой главе. Помимо методов, снабжаться аннотациями могут классы, поля и локальные переменные, а сами аннотации — размещаться на тех же местах, где и модификаторы типа `public` или `static`. И как будет показано в разделе 8.4, аннотациями можно также снабжать пакеты, переменные параметров, параметры типа и примеры применения типов данных.

Каждая аннотация должна определяться с помощью *интерфейса аннотаций*. Методы такого интерфейса должны соответствовать элементам определяемой им аннотации. Например, аннотация `Test` для модульного тестирования средствами JUnit определяется с помощью такого интерфейса:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test
{
    long timeout() default 0L;
    ...
}
```

В объявлении `@interface` создается конкретный интерфейс Java, а инструментальные средства, обрабатывающие аннотации, получают объекты, классы которых реализуют сам интерфейс аннотаций. Например, для извлечения элемента `timeout` из конкретной аннотации `Test` инструментальное средство вызывает метод `timeout()`.

Аннотации `Target` и `Retention` являются *мета-аннотациями*. Они помечают аннотацию `Target`, превращая ее в такую аннотацию, которая может применяться только к методам и должна сохраняться при загрузке файла класса в виртуальную машину. Более подробно они рассматриваются далее, в разделе 8.5.3.

Итак, в этом разделе были представлены основные понятия метаданных и аннотаций к программам. А в следующем разделе будет рассмотрен конкретный пример обработки аннотаций.

8.3.2. Пример аннотирования обработчиков событий

Одной из наиболее утомительных задач в программировании пользовательских интерфейсов является присоединение приемников к источникам событий. Многие приемники событий имеют следующий вид:

```
myButton.addActionListener(() -> doSomething());
```

Во избежание подобной утомительной работы в этом разделе будет разработана специальная аннотация. Определение этой аннотации приведено в листинге 8.11, а ниже представлена ее общая форма.

```
@ActionListenerFor(source="myButton") void doSomething() { . . . }
```

Разработчику больше не придется делать вызовы метода `addActionListener()`. Вместо этого каждый метод приемника событий помечается соответствующей аннотацией. В листинге 8.10 представлен класс `ButtonFrame`, рассматривавшийся в главе 11 первого тома настоящего издания и реализованный заново вместе с подобными аннотациями. Для этого придется также реализовать интерфейс аннотаций, как показано в листинге 8.11.

Разумеется, сами аннотации ничего не делают. Они просто хранятся в исходном файле. Компилятор размещает их в файле классов, а виртуальная машина загружает их. Следовательно, требуется какой-то механизм для анализа аннотаций и установки приемников действий. Эта обязанность возлагается на класс `ActionListenerInstaller`. В конструкторе класса `ButtonFrame` вызывается следующий метод:

```
ActionListenerInstaller.processAnnotations(this);
```

В статическом методе `processAnnotations()` перечисляются все методы объекта, которые он получает. Для каждого метода извлекается и обрабатывается объект аннотации типа `ActionListenerFor`, как показано ниже.

```
Class<?> cl = obj.getClass();
for (Method m : cl.getDeclaredMethods())
{
    ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
    if (a != null) . . .
}
```

В данном случае применяется метод `getAnnotation()`, определенный в интерфейсе `AnnotatedElement`. Этот интерфейс реализуют такие классы, как `Method`, `Constructor`, `Field`, `Class` и `Package`. Имя исходного поля хранится в объекте аннотации. Оно извлекается в результате вызова метода `source()`, а затем осуществляется поиск соответствующего ему поля, как следует из приведенного ниже примера кода.

```
String fieldName = a.source();
Field f = cl.getDeclaredField(fieldName);
```

Этот пример демонстрирует ограниченность рассматриваемой здесь аннотации. Исходный элемент должен непременно обозначать имя поля, а представлять локальную переменную он не может.

Остальная часть кода из рассматриваемого здесь примера носит довольно технический характер. В частности, для каждого аннотируемого метода создается

прокси-объект, класс которого реализует интерфейс `ActionListener` с методом `actionPerformed()`, вызывающим данный аннотируемый метод. (Подробнее о прокси-объектах см. в главе 6 первого тома настоящего издания.) Не особенно вдаваясь в подробности, которые здесь не так важны, отметим самое главное: функциональные возможности аннотаций устанавливаются с помощью метода `processAnnotations()`. Весь процесс обработки аннотаций схематически представлен на рис. 8.1.

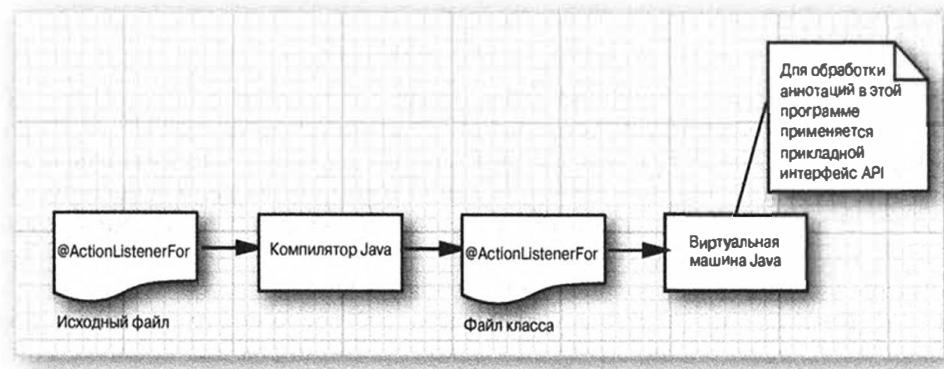


Рис. 8.1. Обработка аннотаций во время выполнения

В рассматриваемом здесь примере аннотации обрабатываются во время выполнения. Но их можно также обрабатывать и на уровне исходного кода. В частности, генератор исходного кода может сгенерировать код для ввода приемников событий. С другой стороны, аннотации можно также обрабатывать на уровне байт-кода. Для этого редактор байт-кода может вставлять вызовы метода `addActionListener()` в конструктор фрейма. На первый взгляд эта задача кажется слишком сложной, но в настоящее время доступны библиотеки, упрощающие ее решение. Удостовериться в этом у вас будет возможность на конкретном примере, представленном далее, в разделе 8.7.

Пример, рассматриваемый в этом разделе, ни в коей мере нельзя рекомендовать в качестве серьезного средства для разработки пользовательских интерфейсов. Применение служебного метода для ввода приемников событий может оказаться не менее удобным, чем вставка аннотаций в исходный код. (На самом деле попытка реализовать именно такой подход предпринята в классе `java.beans.EventHandler`. Этот класс нетрудно сделать действительно полезным, предоставив метод, вводящий обработчик событий, вместо того чтобы создавать его.)

Но данный пример призван продемонстрировать внутренний механизм аннотирования программы и анализа аннотаций. Его рассмотрение должно помочь вам как следует подготовиться к изучению материала последующих разделов, где приводится подробное описание синтаксиса аннотаций.

Листинг 8.9. Исходный код из файла runtimeAnnotations/
ActionListenerInstaller.java

```
1 package runtimeAnnotations;
2
3 import java.awt.event.*;
4 import java.lang.reflect.*;
5
6 /**
7  * @version 1.00 2004-08-17
8  * @author Cay Horstmann
9 */
10 public class ActionListenerInstaller
11 {
12     /**
13      * Обрабатывает все аннотации типа ActionListenerFor
14      * в заданном объекте
15      * @param obj Объект, методы которого могут иметь аннотации
16      *           типа ActionListenerFor
17     */
18     public static void processAnnotations(Object obj)
19     {
20         try
21         {
22             Class<?> cl = obj.getClass();
23             for (Method m : cl.getDeclaredMethods())
24             {
25                 ActionListenerFor a
26                     = m.getAnnotation(ActionListenerFor.class);
27                 if (a != null)
28                 {
29                     Field f = cl.getDeclaredField(a.source());
30                     f.setAccessible(true);
31                     addListener(f.get(obj), obj, m);
32                 }
33             }
34         }
35         catch (ReflectiveOperationException e)
36         {
37             e.printStackTrace();
38         }
39     }
40
41     /**
42      * Вводит приемник действий, вызывающий заданный метод
43      * @param source Источник событий, в который вводится
44      *               приемник действий
45      * @param param Неявный параметр метода, вызываемого
46      *               приемником действий
47      * @param m Метод, вызываемый приемником действий
48     */
49     public static void addListener(Object source,
50                                     final Object param, final Method m)
51                                     throws ReflectiveOperationException
52     {
53         InvocationHandler handler = new InvocationHandler()
54         {
55             public Object invoke(Object proxy, Method mm,
56                                 Object[] args)
```

```
57             throws Throwable
58         {
59             return m.invoke(param);
60         }
61     };
62
63     Object listener = Proxy.newProxyInstance(null, new Class[]
64         { java.awt.event.ActionListener.class }, handler);
65     Method adder = source.getClass().getMethod(
66         "addActionListener", ActionListener.class);
67     adder.invoke(source, listener);
68 }
69 }
```

Листинг 8.10. Исходный код из файла buttons3/ButtonFrame.java

```
1 package buttons3;
2
3 import java.awt.*;
4 import javax.swing.*;
5 import runtimeAnnotations.*;
6
7 /**
8  * Фрейм с панелью кнопок
9  * @version 1.00 2004-08-17
10 * @author Cay Horstmann
11 */
12 public class ButtonFrame extends JFrame
13 {
14     private static final int DEFAULT_WIDTH = 300;
15     private static final int DEFAULT_HEIGHT = 200;
16
17     private JPanel panel;
18     private JButton yellowButton;
19     private JButton blueButton;
20     private JButton redButton;
21
22     public ButtonFrame()
23     {
24         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
25
26         panel = new JPanel();
27         add(panel);
28         yellowButton = new JButton("Yellow");
29         blueButton = new JButton("Blue");
30         redButton = new JButton("Red");
31
32         panel.add(yellowButton);
33         panel.add(blueButton);
34         panel.add(redButton);
35
36         ActionListenerInstaller.processAnnotations(this);
37     }
38
39     @ActionListenerFor(source = "yellowButton")
40     public void yellowBackground()
41     {
42         panel.setBackground(Color.YELLOW);
43     }
44 }
```

```
43 }
44
45 @ActionListenerFor(source = "blueButton")
46 public void blueBackground()
47 {
48     panel.setBackground(Color.BLUE);
49 }
50
51 @ActionListenerFor(source = "redButton")
52 public void redBackground()
53 {
54     panel.setBackground(Color.RED);
55 }
56 }
```

Листинг 8.11. Исходный код из файла runtimeAnnotations/ActionListenerFor.java

```
1 package runtimeAnnotations;
2
3 import java.lang.annotation.*;
4
5 /**
6  * @version 1.00 2004-08-17
7  * @author Cay Horstmann
8 */
9
10 @Target(ElementType.METHOD)
11 @Retention(RetentionPolicy.RUNTIME)
12 public @interface ActionListenerFor
13 {
14     String source();
15 }
```

java.lang.reflect.AnnotatedElement 5.0

- **boolean isAnnotationPresent(Class<? extends Annotation> annotationType)**
Возвращает логическое значение **true**, если данный элемент имеет аннотацию указанного типа.
- **<T extends Annotation> T getAnnotation(Class<T> annotationType)**
Получает аннотацию заданного типа или пустое значение **null**, если такая аннотация отсутствует у данного элемента.
- **<T extends Annotation> T[] getAnnotationsByType(Class<T> annotationType) 8**
Получает все аннотации повторяющегося типа (см. далее, раздел 8.5.3) или возвращает массив нулевой длины.
- **Annotation[] getAnnotations()**
Получает все аннотации, доступные для данного элемента, включая унаследованные. Если доступные аннотации отсутствуют, возвращает массив нулевой длины.
- **Annotation[] getDeclaredAnnotations()**
Получает все аннотации, объявленные для данного элемента, исключая унаследованные. Если доступные аннотации отсутствуют, возвращает массив нулевой длины.

8.4. Синтаксис аннотаций

В последующих разделах поясняется все, что следует знать о синтаксисе аннотаций.

8.4.1. Интерфейсы аннотаций

Аннотация определяется с помощью своего интерфейса следующим образом:

```
модификаторы @interface ИмяАннотации
{
    объявлениеЭлемента1
    объявлениеЭлемента2
    ...
}
```

Каждый элемент может быть объявлен в одной из следующих форм:

тип имяЭлемента();

или

тип имяЭлемента() default значение;

Например, приведенная ниже аннотация состоит из двух элементов: assignedTo и severity.

Например, приведенная ниже аннотация состоит из двух элементов: assignedTo и severity.

```
public @interface BugReport
{
    String assignedTo() default "[none]";
    int severity();
}
```

Все интерфейсы аннотаций неявно расширяют интерфейс `java.lang.annotation.Annotation`. Это обычный интерфейс, а не интерфейс аннотаций. Представляемые им методы можно найти в конце этого раздела, где описывается прикладной программный интерфейс API. Самостоятельно расширять интерфейсы аннотаций нельзя. Иными словами, все интерфейсы аннотаций могут только расширять непосредственно интерфейс `java.lang.annotation.Annotation`. Нельзя также предоставить классы, реализующие интерфейсы аннотаций.

Методы из интерфейса аннотаций могут не иметь ни параметров, ни операторов `throws`. Они не могут быть статическими, обобщенными с параметрами типа или методами по умолчанию.

Элемент аннотации может принимать один из следующих типов.

- Примитивный тип (`int`, `short`, `long`, `byte`, `char`, `double`, `float` или `boolean`).
- Строковый тип `String`.
- Тип `Class` (с каким-нибудь необязательным параметром вроде `Class<? extends MyClass>`).
- Перечислимый тип `enum`.
- Тип аннотации.

- Массив перечисленных выше типов (массив массивов не является допустимым типом элемента аннотации).

Ниже приведены примеры объявления элементов аннотации, имеющих допустимые типы.

```
public @interface BugReport
{
    enum Status { UNCONFIRMED, CONFIRMED, FIXED, NOTABUG };
    boolean showStopper() default false;
    String assignedTo() default "[none]";
    Class<?> testCase() default Void.class;
    Status status() default Status.UNCONFIRMED;
    Reference ref() default @Reference(); // тип аннотации
    String[] reportedBy();
}
```

`java.lang.annotation.Annotation` 5.0

- `Class<? extends Annotation> annotationType()`

Возвращает объект типа `Class`, который представляет интерфейс аннотаций данного объекта аннотации. Следует, однако, иметь в виду, что вызов метода `getClass()` для объекта аннотации привел бы к возврату конкретного класса, а не интерфейса.

- `boolean equals(Object other)`

Возвращает логическое значение `true`, если параметр `other` обозначает объект, класс которого реализует тот же интерфейс аннотаций, что и данный объект аннотации, и если все элементы этого объекта равны параметру `other`.

- `int hashCode()`

Возвращает хеш-код, совместимый с методом `equals()` и получаемый из имени интерфейса аннотаций и значений его элементов.

- `String toString()`

Возвращает строковое представление, содержащее название интерфейса аннотаций и значения элементов, например `@BugReport(assignedTo=[none], severity=0)`.

8.4.2. Объявление аннотаций

Каждая аннотация имеет следующий формат:

`@ИмяАннотации(имяЭлемента1=значение1, имяЭлемента2=значение2, . . .)`

В качестве примера ниже приведено объявление аннотации.

`@BugReport(assignedTo="Harry", severity=10)`

Порядок следования элементов в аннотации особого значения не имеет. Так, следующая аннотация идентична приведенной выше:

`@BugReport(severity=10, assignedTo="Harry")`

Значение, указываемое в объявлении аннотации по умолчанию, используется в том случае, если для элемента аннотации не задано конкретное значение. Ниже приведен пример аннотации, где в качестве значения для элемента `assignedTo` выбирается символьная строка "[поле]".

`@BugReport(severity=10)`



Внимание! Устанавливаемые по умолчанию значения не хранятся вместе с аннотацией, а вычисляются динамически. Так, если заменить значение, устанавливаемое по умолчанию в элементе `assignedTo`, строковым значением "[]" и скомпилировать интерфейс `BugReport` заново, в аннотации `@BugReport(severity=10)` будет использоваться новое устанавливаемое по умолчанию значение — даже в тех файлах классов, которые были скомпилированы до изменения этого значения.

Аннотации можно упростить с помощью двух специальных сокращений. Так, если элементы не указаны, потому что они просто отсутствуют в аннотации или все они принимают значения, устанавливаемые по умолчанию, то круглые скобки не требуются. Например, аннотация

```
@BugReport
```

равнозначна аннотации

```
@BugReport(assignedTo=" [none] ", severity=0)
```

Подобная аннотация называется *маркерной*.

Другим специальным сокращением является *однозначная аннотация*. Если элемент аннотации имеет специальное имя `value` и больше никаких элементов не указано, то имя элемента и знак = могут быть опущены. Так, если определить интерфейс аннотаций `ActionListenerFor`, упоминавшийся в предыдущем разделе, следующим образом:

```
public @interface ActionListenerFor
{
    String value();
}
```

то сами аннотации можно записать так:

```
@ActionListenerFor("yellowButton")
```

а не так, как показано ниже.

```
@ActionListenerFor(value="yellowButton")
```

У каждого элемента может быть несколько аннотаций, как показано ниже.

```
@BugReport(showStopper=true, reportedBy="Joe")
@BugReport(reportedBy={"Harry", "Carl"})
void myMethod()
```

Если автор аннотации объявил ее повторяющейся, такую аннотацию можно повторять неоднократно, как демонстрируется в следующем примере кода:

```
@BugReport(showStopper=true, reportedBy="Joe")
@BugReport(reportedBy={"Harry", "Carl"})
public void checkRandomInsertions()
```



На заметку! Аннотации вычисляются компилятором, и поэтому значения всех элементов аннотаций должны быть представлены константами, обрабатываемыми во время компиляции, как показано в следующем примере кода:

```
@BugReport(showStopper=true, assignedTo="Harry",
           testCase=MyTestCase.class,
           status=BugReport.Status.CONFIRMED, . . .)
```



Внимание! Элемент аннотации вообще не может принимать пустое значение `null`. Даже по умолчанию не допускается устанавливать пустое значение `null`. Это не очень удобно, поскольку для установки по умолчанию придется выбирать другие значения вроде "" или `Void.class`.

Если в качестве значения элемента аннотации указывается массив, значения элементов такого массива заключаются в фигурные скобки:

```
@BugReport(. . ., reportedBy={"Harry", "Carl"})
```

Если же элемент аннотации принимает единственное значение, фигурные скобки можно опустить:

```
@BugReport(. . ., reportedBy="Joe") // допускается и подобно {"Joe"}
```

В качестве элемента аннотации может служить какая-нибудь другая аннотация. Это дает возможность создавать довольно сложные аннотации, как показано в приведенном ниже примере.

```
@BugReport(ref=@Reference(id="3352627"), . . .)
```



На заметку! Внедрение циклических зависимостей в аннотациях считается ошибкой. Например, вследствие того, что аннотация `BugReport` содержит элемент типа `Reference`, аннотация `Reference` не может содержать элемент типа `BugReport`.

8.4.3. Аннотирование объявлений

Аннотации могут встречаться и во многих других местах прикладного кода. Эти места можно разделить на две категории: *объявления* и *места употребления* типов данных. Аннотации могут появляться в объявлениях следующих элементов кода.

- Пакеты.
- Классы (включая и перечисления).
- Методы.
- Конструкторы.
- Переменные экземпляра (включая и константы перечислимого типа).
- Локальные переменные.
- Переменные параметров.
- Параметры типа.

В объявлениях классов и интерфейсов аннотации указываются перед ключевым словом `class` или `interface` следующим образом:

```
@Entity public class User { ... }
```

А в объявлениях переменных аннотации указываются перед типом переменной таким образом:

```
@SuppressWarnings("unchecked") List<User> users = ...;
public User getUser(@Param("id") String userId)
```

Параметр типа в обобщенном классе или методе может быть аннотирован следующим образом:

```
public class Cache<@Immutable V> { ... }
```

Пакет аннотируется в отдельном файле package-info.java. Этот файл содержит только операторы объявления и импорта пакета с предшествующими аннотациями, как показано ниже. Обратите внимание на то, что оператор import следует *после* оператора package, в котором объявляется пакет.

```
/**  
 * Документирующий комментарий на уровне пакета  
 */  
@GPL(version="3")  
package com.horstmann.corejava;  
import org.gnu.GPL;
```



На заметку! Аннотации всех локальных переменных отбрасываются при компилировании класса. Следовательно, они могут быть обработаны только на уровне исходного кода. Аналогично аннотации пакетов не сохраняются вне уровня исходного кода.

8.4.4. Аннотирование в местах употребления типов данных

Аннотация в объявлении предоставляет некоторые сведения об объявляемом элементе кода. Так, в следующем примере кода аннотацией утверждается, что параметр userId объявляемого метода не является пустым:

```
public User getUser(@NotNull String userId)
```



На заметку! Аннотация @NotNull является частью каркаса Checker Framework (<http://types.cs.washington.edu/checker-framework>). С помощью этого каркаса можно включать утверждения в прикладную программу, например, утверждение, что параметр не является пустым или относится к типу **String** и содержит регулярное выражение. В таком случае инструментальное средство статистического анализа проверит достоверность утверждений в данном теле исходного кода.

А теперь допустим, что имеется параметр типа `List<String>` и требуется каким-то образом указать, что все символьные строки не являются пустыми. Именно здесь и пригодятся аннотации в местах употребления типов. Такую аннотацию достаточно указать перед аргументом типа следующим образом:

```
List<@NotNull String>
```

Подобные аннотации можно указывать в следующих местах употребления типов данных.

- Вместе с аргументами обобщенного типа: `List<@NotNull String>, Comparator.<@NotNull String> reverseOrder()`.
- В любом месте массива: `@NotNull String[][] words` (элемент массива `words[i][j]` не является пустым), `String @NotNull [][] words` (массив `words` не является пустым), `String[] @NotNull [] words` (элемент массива `words[i]` не является пустым).
- В суперклассах и реализуемых интерфейсах: `class Warning extends @Localized Message`.
- В вызовах конструкторов: `new @Localized String(...)`.
- Во вложенных типах: `Map.<@Localized Entry>`.

- В операции приведения и проверки типов instanceof: (@Localized String) text, if (text instanceof @Localized String). (Аннотации служат для употребления только внешними инструментальными средствами. Они не оказывают никакого влияния на поведение операции приведения и проверки типов instanceof.)
- В местах указания исключений: public String read() throws @Localized IOException.
- Вместе с метасимволами подстановки и ограничениями типов: List<@Localized ? extends Message>, List<? Extends @Localized Message>.
- В ссылках на методы и конструкторы: @Localized Message::getText.

Имеются все же некоторые методы употребления типов, где аннотации не допускаются. Ниже приведены характерные тому примеры.

```
@NonNull String.class // ОШИБКА: литерал класса не
                        // подлежит аннотированию!
import java.lang.@NonNull String; // ОШИБКА: импорт не
                                    // подлежит аннотированию!
```

Аннотации можно размещать до или после других модификаторов доступа вроде private и static. Обычно (хотя и не обязательно) аннотации в местах употребления типов размещаются после других модификаторов доступа, тогда как аннотации в объявлениях – перед другими модификаторами доступа. Ниже приведены характерные тому примеры.

```
private @NonNull String text; // Аннотация в месте употребления типа
@Id private String userId; // Аннотация в объявлении переменной
```



На заметку! Автор аннотации должен указать место, в котором может появиться конкретная аннотация. Если аннотация допускается как в объявлении переменной, так и в месте употребления типа, а применяется в объявлении переменной, то она указывается и в том и в другом месте. Рассмотрим в качестве примера следующее объявление метода:

```
public User getUser(@NonNull String userId)
```

Если аннотацию @NonNull можно применять как в параметрах, так и в местах употребления типов, то параметр userId аннотируется, а тип параметра обозначается как @NonNull String.

8.4.5. Аннотирование по ссылке this

Допустим, требуется аннотировать параметры, которые не изменяются методом:

```
public class Point {
    public boolean equals(@ReadOnly Object other) { ... }
}
```

В таком случае инструментальное средство, обрабатывающее данную аннотацию, после анализа следующего вызова:

```
p.equals(q)
```

посчитает, что параметр q не изменился. А как насчет ссылки p? При вызове данного метода переменная this привязывается к ссылке p. Но ведь переменная this вообще не объявляется, а следовательно, она и не может быть аннотирована.

На самом деле эту переменную можно объявить с помощью редко употребляемой разновидности синтаксиса, чтобы ввести аннотацию следующим образом:

```
public class Point {public boolean equals(
    @ReadOnly Point this,
    @ReadOnly Object other) { ... }
}
```

Первый параметр в приведенном выше примере кода называется *параметром получателя*. Он должен непременно называться **this**. Его тип относится к тому классу, объект которого создается.



На заметку! Параметром получателя можно снабдить только методы, но не конструкторы. По существу, ссылка **this** в конструкторе не является объектом данного типа до тех пор, пока конструктор не завершится. Напротив, аннотация, размещаемая в конструкторе, описывает конструируемый объект.

Конструктору внутреннего класса передается другой скрытый параметр, а именно: ссылка на объект объемлющего класса. Этот параметр также можно указать явным образом:

```
static class Sequence {
    private int from;
    private int to;

    class Iterator implements java.util.Iterator<Integer> {
        private int current;

        public Iterator(@ReadOnly Sequence Sequence.this) {
            this.current = Sequence.this.from;
        }
        ...
    }
    ...
}
```

Этот параметр именуется таким же образом, как и при ссылке на него: *ОбъемлющийКласс.this*. А его тип относится к объемлющему классу.

8.5. Стандартные аннотации

В версии Java SE имеется ряд интерфейсов аннотаций, определенных в таких пакетах, как `java.lang`, `java.lang.annotation` и `javax.annotation`. Четыре из них являются мета-аннотациями, описывающими поведение интерфейсов аннотаций, а остальные — обычными аннотациями, которые разработчики могут применять для аннотирования элементов в своем исходном коде. Все эти аннотации вкратце перечислены в табл. 8.2 и более подробно будут описаны в двух последующих разделах.

Таблица 8.2. Стандартные аннотации

Интерфейс аннотаций	Применение	Назначение
Deprecated	Все элементы кода	Помечает элемент кода как не рекомендуемый для применения
SuppressWarnings	Все элементы кода, кроме пакетов и аннотаций	Подавляет предупреждения указанного типа
SafeVarargs	Методы и конструкторы	Утверждает, что аргументы переменной длины безопасны для употребления
Override	Методы	Проверяет, переопределяет ли данный метод соответствующий метод из суперкласса
FunctionalInterface	Интерфейсы	Обозначает интерфейс как функциональный с единственным абстрактным методом
PostConstruct, PreDestroy	Методы	Помеченный метод должен вызываться сразу же после создания или непосредственно перед удалением
Resource	Классы, интерфейсы, методы, поля	Если это класс или интерфейс, помечает его как ресурс для использования в каком-нибудь другом месте. А если это метод или поле, то помечает его как ресурс для "внедрения"
Resources	Классы, интерфейсы	Помечает массив ресурсов
Generated	Все элементы кода	Помечает элемент как исходный код, сгенерированный каким-нибудь инструментальным средством
Target	Аннотации	Обозначает элементы, к которым может быть применена данная аннотация
Retention	Аннотации	Обозначает, как долго должна сохраняться данная аннотация
Documented	Аннотации	Обозначает, что данная аннотация должна быть включена в документацию на аннотируемые элементы кода
Inherited	Аннотации	Обозначает, что если данная аннотация применяется к классу, она будет автоматически наследоваться всеми его подклассами
Repeatable	Аннотации	Обозначает, что данную аннотацию можно неоднократно применять к одному и тому же элементу

8.5.1. Аннотации для компиляции

Аннотация `@Deprecated` может присоединяться к любым элементам, применение которых впредь не рекомендуется. В таком случае компилятор будет выдавать соответствующее предупреждение, если нерекомендуемый элемент все-таки применяется в исходном коде. Эта аннотация имеет то же назначение, что и дескриптор `@deprecated` в Javadoc.

Аннотация `@SuppressWarnings` указывает компилятору подавлять предупреждения определенного типа, как показано в приведенном ниже примере.

```
@SuppressWarnings("unchecked")
```

Аннотация `@Override` применяется только к методам. Компилятор проверяет, чтобы метод с такой аннотацией действительно переопределял соответствующий метод из суперкласса. Так, если сделать приведенное ниже объявление, компилятор выдаст ошибку, поскольку метод `equals()` не переопределяет аналогичный метод `equals()` из класса `Object`. Этот метод имеет параметр типа `Object`, а не `MyClass`.

```
public MyClass
{
    @Override public boolean equals(MyClass other);
    . . .
}
```

Аннотация `@Generated` предназначена для использования инструментальными средствами генерирования кода. Любой генерируемый исходный код может снабжаться аннотациями, чтобы отличаться от кода, предоставляемого программистом. Например, редактор кода может скрывать сгенерированный код, а генератор кода — удалять более старые версии сгенерированного кода. Каждая такая аннотация должна содержать уникальный идентификатор для генератора кода. Символьные строки даты (в формате ISO 8601) и комментариев указывать необязательно. Ниже приведен характерный пример аннотации для генерирования кода.

```
@Generated("com.horstmann.beanproperty",
           "2008-01-04T12:08:56.235-0700");
```

8.5.2. Аннотации для управления ресурсами

Аннотации `@PostConstruct` и `@PreDestroy` применяются в таких средах, которые управляют жизненным циклом объектов, например, в веб-контейнерах и серверах приложений. Методы, помечаемые такими аннотациями, должны вызываться сразу же после создания объекта или непосредственно перед его удалением.

Аннотация `@Resource` предназначена для внедрения ресурсов. Рассмотрим в качестве примера веб-приложение, получающее доступ к базе данных. Разумеется, сведения о получении доступа к базе данных не должны жестко кодироваться в таком приложении. Вместо этого у веб-контейнера должен быть какой-то пользовательский интерфейс для установки параметров подключения, а также имя JNDI для источника данных. Сослаться на источник данных в веб-приложении можно следующим образом:

```
@Resource(name="jdbc/mydb")
private DataSource source;
```

8.5.3. Мета-аннотации

Мета-аннотация `@Target` применяется к аннотации, ограничивая те элементы, которые должны быть снабжены данной аннотацией, как показано в приведенном ниже примере.

```
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface BugReport
```

В табл. 8.3 перечислены все возможные значения элементов данной мета-аннотации. Они относятся к перечислимому типу ElementType. Указывать можно любое количество типов элементов, заключая их в фигурные скобки.

Таблица 8.3. Типы элементов для мета-аннотации @Target

Тип элемента	Применение аннотации
ANNOTATION_TYPE	Объявления типов аннотаций
PACKAGE	Пакеты
TYPE	Классы (включая перечисления) и интерфейсы (включая типы аннотаций)
METHOD	Методы
CONSTRUCTOR	Конструкторы
FIELD	Поля (включая константы перечислимого типа)
PARAMETER	Параметры методов или конструкторов
LOCAL_VARIABLE	Локальные переменные
TYPE_PARAMETER	Параметры типа
TYPE_USE	Примеры применения типов данных

Аннотацией без ограничений, накладываемых мета-аннотацией @Target, можно снабдить любой элемент кода. Компилятор проверяет, чтобы аннотацией снабжался только разрешенный элемент кода. Так, если снабдить аннотацией @BugReport поле, компилятор выдаст во время компиляции соответствующую ошибку.

Мета-аннотация @Retention обозначает, как долго должна сохраняться аннотация. Указать можно не больше одного значения из тех, что приведены в табл. 8.4. По умолчанию устанавливается значение константы RetentionPolicy.CLASS.

Таблица 8.4. Правила сохраняемости для мета-аннотации @Retention

Правило сохраняемости	Описание
SOURCE	Аннотации не включаются в файлы классов
CLASS	Аннотации включаются в файлы классов, но виртуальной машине не нужно их загружать
RUNTIME	Аннотации включаются в файлы классов и загружаются виртуальной машиной. Они доступны через прикладной интерфейс API для рефлексии

В упоминавшемся ранее примере реализации интерфейса аннотаций из листинга 8.11 аннотация @ActionListenerFor была объявлена со значением константы RetentionPolicy.RUNTIME, поскольку для обработки аннотаций в данном примере применялась рефлексия. В двух последующих разделах будут приведены примеры обработки аннотаций как на уровне исходного кода, так и на уровне файлов классов.

Мета-аннотация @Documented выдает подсказку о средствах документирования вроде Javadoc. Документируемые аннотации должны применяться в целях документирования таким же образом, как и другие модификаторы, подобные protected или static. О применении остальных аннотаций в документации не упоминается. Допустим, аннотация @ActionListenerFor объявляется как документируемая следующим образом:

```
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ActionListenerFor
```

В таком случае документация на каждый аннотированный метод будет содержать эту аннотацию (рис. 8.2). Если же аннотация оказывается временной (как, например, @BugReport), то документировать ее применение вряд ли стоит.

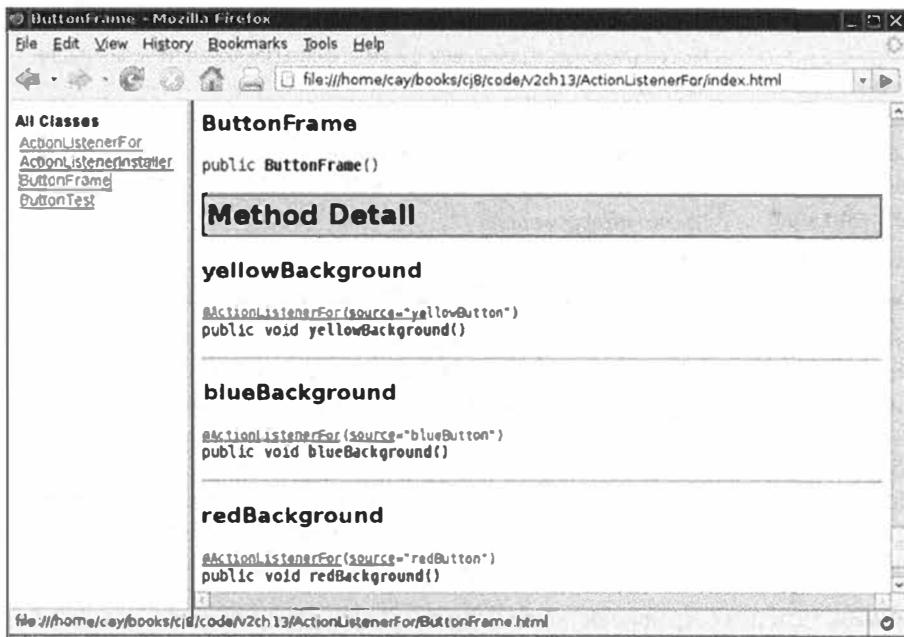


Рис. 8.2. Документируемые аннотации

На заметку! Аннотацию вполне допустимо применять к самой себе. Например, аннотация `@Documented` аннотируется как `@Documented`. Поэтому в документации Javadoc на аннотации указано, являются ли они документируемыми.

Мета-аннотация `@Inherited` применяется только к аннотациям классов. Когда в класс вводится наследуемая аннотация, все его подклассы автоматически снабжаются точно такой же аннотацией. Это позволяет легко создавать аннотации, действующие таким же образом, как и маркерные интерфейсы вроде `Serializable`.

На самом деле аннотация `@Serializable` будет более уместной, чем маркерный интерфейс `Serializable` без единого метода. Класс является сериализируемым из-за наличия во время выполнения поддержки для чтения и записи его полей, а не из-за каких-то принципов объектно-ориентированного проектирования. Аннотация описывает данное обстоятельство намного лучше, чем наследование интерфейсов, ведь интерфейс `Serializable` появился еще в версии JDK 1.1, т.е. много раньше, чем аннотации.

Допустим, для указания на то, что объекты класса могут сохраняться в базе данных, определяется наследуемая аннотация `@Persistent`. В таком случае все подклассы этого класса будут автоматически аннотироваться как сохраняемые следующим образом:

```
@Inherited @interface Persistent { }
@Persistent class Employee { . . . }
class Manager extends Employee { . . . } // также @Persistent
```

При поиске объектов, сохраняемых в базе данных, механизм сохраняемости будет обнаруживать как объекты типа `Employee`, так и объекты типа `Manager`.

Начиная с версии Java SE 8 допускается неоднократное применение аннотации одного и того же типа к отдельному элементу. Ради обратной совместимости разработчикам повторяющейся аннотации пришлось предоставить *контейнерную аннотацию*, содержащую повторяющиеся аннотации в массиве. Ниже показано, каким образом определяются аннотация `@TestCase` и ее контейнер.

```
@Repeatable(TestCases.class)
@interface TestCase {
    String params();
    String expected();
}

@interface TestCases {
    TestCase[] value();
}
```

Всякий раз, когда пользователь предоставляет две или больше аннотации `@TestCase`, они автоматически заключаются в оболочку аннотации `@TestCases`.



Внимание! Обработка повторяющихся аннотаций требует особого внимания. Если вызвать метод `getAnnotation()` для поиска повторяющейся аннотации, которая на самом деле не повторялась, то и в этом случае может быть получено пустое значение `null`. Объясняется это тем, что повторяющиеся аннотации были заключены в оболочку контейнерной аннотации.

В таком случае следует вызвать метод `getAnnotationsByType()`, где просматривается контейнер и предоставляется массив повторяющихся аннотаций. Если бы имелась только одна аннотация, она была бы получена в массиве единичной длины. Имея в своем распоряжении данный метод, можно вообще не беспокоиться о контейнерной аннотации.

8.6. Обработка аннотаций на уровне исходного кода

В предыдущем разделе было показано, каким образом аннотации анализируются в выполняющейся программе. Еще одним примером применения аннотаций служит автоматическая обработка исходных файлов для получения дополнительного исходного кода, файлов конфигурации, сценариев и вообще всего, что можно генерировать.

8.6.1. Процессоры аннотаций

Обработка аннотаций встроена в компилятор Java. Во время компиляции *процессоры аннотаций* можно вызывать по следующей команде:

```
javac -processor ИмяКлассаПроцессора1, ИмяКлассаПроцессора2, ...
    Исходные_файлы
```

Компилятор обнаруживает аннотации в исходных файлах. Каждый процессор аннотаций выполняется по очереди с учетом тех аннотаций, к которым он проявил интерес. Если процессор аннотаций создает новый исходный файл, то данный процесс повторяется. Как только все исходные файлы будут обработаны, они компилируются.



На заметку! Процессор аннотаций может только формировать новые исходные файлы и не может изменять уже имеющиеся исходные файлы.

Процессор аннотаций реализует интерфейс `Processor`, как правило, расширяя класс `AbstractProcessor`. При этом нужно указать, какие именно аннотации поддерживаются процессором. В данном случае это следующие аннотации:

```
@SupportedAnnotationTypes("com.horstmann.annotations.ToString")
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class ToStringAnnotationProcessor extends
    AbstractProcessor
{
    @Override
    public boolean process(Set<? extends TypeElement> annotations,
        RoundEnvironment currentRound)
    {
        ...
    }
}
```

Процессору могут потребоваться конкретные типы аннотаций, метасимволы подстановки вроде "com.horstmann.*" (т.е. все аннотации из пакета com.horstmann и любых его подпакетов) или даже "*" (т.е. все аннотации вообще). Метод `process()` вызывается один раз на каждом цикле обработки со всеми аннотациями, обнаруженными в любых файлах в данном цикле, а также со ссылкой на интерфейс `RoundEnvironment`, содержащей сведения о текущем цикле обработки.

8.6.2. Прикладной программный интерфейс API модели языка

Для анализа аннотаций на уровне исходного кода служит прикладной программный интерфейс API модели языка. В отличие от прикладного программного интерфейса API для рефлексии, представляющего классы и методы на уровне виртуальной машины, прикладной программный интерфейс API модели языка позволяет анализировать программу на Java по правилам языка Java.

Компилятор получает дерево, узлами которого являются экземпляры классов, реализующих интерфейс `javax.lang.model.element.Element` и производные от него интерфейсы `TypeElement`, `VariableElement`, `ExecutableElement` и т.д. Они служат статическими аналогами классов рефлексии `Class`, `Field/Parameter`, `Method/Constructor`.

Не вдаваясь в подробности прикладного программного интерфейса API модели языка, ниже перечислены главные его особенности, о которых нужно знать для обработки аннотаций.

- Интерфейс RoundEnvironment предоставляет все элементы кода, помеченные конкретной аннотацией. Для этой цели вызывается следующий метод:
`Set<? extends Element> getElementsAnnotatedWith(
 Class<? extends Annotation> a)`

- Эквивалентом интерфейса AnnotateElement для обработки аннотаций на уровне исходного кода является интерфейс AnnotatedConstruct. Для получения обычных или повторяющихся аннотаций из отдельного аннотированного класса служат следующие методы:

```
A getAnnotation(Class<A> annotationType)  
A[] getAnnotationsByType(Class<A> annotationType)
```

- Интерфейс TypeElement представляет класс или интерфейс. А метод `getEnclosedElements()` получает список его полей и методов.
- В результате вызова метода `getSimpleName()` по ссылке типа Element или метода `getQualifiedName()` по ссылке типа TypeElement получается объект типа Name, который может быть преобразован в символьную строку методом `toString()`.

8.6.3. Генерирование исходного кода с помощью аннотаций

В качестве примера рассмотрим применение аннотаций с целью упростить реализацию методов типа `toString`. Такие методы нельзя ввести в исходные классы. Ведь процессоры аннотаций способны производить только новые классы, а не изменять уже имеющиеся. Следовательно, все методы должны быть введены в служебный класс `ToStrings` следующим образом:

```
public class ToStrings {  
    public static String toString(Point obj) {  
        Сгенерированный код  
    }  
    public static String toString(Rectangle obj) {  
        Сгенерированный код  
    }  
    ...  
    public static String toString(Object obj) {  
        return Objects.toString(obj);  
    }  
}
```

В данном случае применять рефлексию не требуется, и поэтому аннотируются методы доступа, но не поля:

```
@ToString  
public class Rectangle {  
    ...  
    @ToString(includeName=false) public Point getTopLeft()  
        { return topLeft; }  
    @ToString public int getWidth() { return width; }  
    @ToString public int getHeight() { return height; }  
}
```

И тогда процессор аннотаций должен генерировать следующий исходный код:

```
public static String toString(Rectangle obj) {
    StringBuilder result = new StringBuilder();
    result.append("Rectangle");
    result.append("[");
    result.append(toString(obj.getTopLeft()));
    result.append(",");
    result.append("width=");
    result.append(toString(obj.getWidth()));
    result.append(",");
    result.append("height=");
    result.append(toString(obj.getHeight()));
    result.append("]");
    return result.toString();
}
```

Шаблонный код выделен выше обычным шрифтом. Ниже приведен набросок метода, получающего метод `toString()` для класса с заданным параметром типа `TypeElement`.

```
private void writeToStringMethod(PrintWriter out, TypeElement te) {
    String className = te.getQualifiedName().toString();
    Вывести заголовок метода и объявление построителя символьных строк
    ToString ann = te.getAnnotation(ToString.class);
    if (ann.includeName())
        Вывести код для ввода имени класса
    for (Element c : te.getEnclosedElements()) {
        ann = c.getAnnotation(ToString.class);
        if (ann != null) {
            if (ann.includeName()) Вывести код для ввода имени поля
                Вывести код для присоединения
                метода toString(obj.ИмяМетода())
        }
    }
    Вывести код для возврата символьной строки
}
```

А ниже приведен набросок метода `process()` из процессора аннотаций. В этом методе создается исходный файл для вспомогательного класса, а также выводится заголовок класса и по одному методу для каждого аннотируемого класса.

```
public boolean process(Set<? extends TypeElement> annotations,
    RoundEnvironment currentRound) {
    if (annotations.size() == 0) return true;
    try {
        JavaFileObject sourceFile =
            processingEnv.getFiler().createSourceFile(
                "com.horstmann.annotations.ToStrings");
        try (PrintWriter out = new PrintWriter(sourceFile.openWriter()))
        {
            Вывести код для пакета и класса
            for (Element e : currentRound.getElementsAnnotatedWith(
                ToString.class))
            {
```

```

        if (e instanceof TypeElement)
        {
            TypeElement te = (TypeElement) e;
            writeToStringMethod(out, te);
        }
    }
    Вывести код для метода toString(Object)
} catch (IOException ex)
{
    processingEnv.getMessager().printMessage(
        Kind.ERROR, ex.getMessage());
}
}
return true;
}

```

За более подробными сведениями обращайтесь к примерам кода, сопровождающего данную книгу. Следует, однако, иметь в виду, что метод `process()` вызывается в последующих циклах обработки аннотаций с пустым списком аннотаций. И тогда происходит немедленный возврат из данного метода, чтобы не создавать исходный файл дважды.

Сначала скомпилируйте процессор аннотаций, а затем скомпилируйте и выполните тестовую программу, введя следующие команды:

```

javac sourceAnnotations/ToStringAnnotationProcessor.java
javac -processor sourceAnnotations.ToStringAnnotationProcessor
      rect/*.java
java rect.SourceLevelAnnotationDemo

```



Совет! Чтобы просмотреть циклы обработки аннотаций, выполните команду `javac` с параметром `-XprintRounds`. В итоге на экран будет выведен результат, аналогичный следующему:

```

Round 1:
input files: {ch11.sec05.Point, ch11.sec05.Rectangle,
              ch11.sec05.SourceLevelAnnotationDemo}
annotations: [com.horstmann.annotations.ToString]
last round: false
Round 2:
input files: {com.horstmann.annotations.ToStrings}
annotations: []
last round: false
Round 3:
input files: {}
annotations: []
last round: true

```

В данном примере было продемонстрировано, каким образом инструментальные средства могут собирать аннотации из исходных файлов для получения других файлов. Формируемые в итоге файлы совсем не обязательно должны быть исходными. Процессоры аннотаций могут сформировать дескрипторы XML-разметки, файлы свойств, сценарии командного процессора, документацию в формате HTML и пр.



На заметку! Были предложения использовать аннотации, чтобы еще больше сократить объем рутинной работы. В самом деле, было бы замечательно, если бы тривиальные методы получения и установки генерировались автоматически. Например, следующая аннотация:

```
@Property private String title;
```

могла бы автоматически генерировать приведенные ниже методы.

```
public String getTitle() { return title; }
public void setTitle(String title) { this.title = title; }
```

Но эти методы должны быть введены в один и тот же класс. Для этого потребуется редактирование исходного файла, а не только генерирование еще одного файла, что выходит за пределы возможностей обработчиков аннотаций. С этой целью можно было бы создать другое инструментальное средство, но оно уже не вписывалось бы в рамки основного назначения аннотаций. Ведь аннотация предназначена для описания элемента кода, а не в качестве директивы для добавления или изменения кода.

8.7. Конструирование байт-кодов

Как пояснялось ранее, аннотации могут обрабатываться во время выполнения и на уровне исходного кода. Но существует еще и третий способ обработки аннотаций на уровне байт-кода. Если аннотации не удаляются на уровне исходного кода, они присутствуют в файлах классов. Формат этих файлов документирован (см. <http://docs.oracle.com/javase/specs/jvms/se8/html>). Это довольно сложный формат, и поэтому обрабатывать файлы классов без специальных библиотек было бы совсем не просто. К их числу относится библиотека ASM, доступная по адресу <http://asm.ow2.org>.

8.7.1. Видоизменение файлов классов

В этом разделе будет показано, как пользоваться библиотекой ASM для добавления в аннотированные методы протокольных сообщений. Так, если метод снабжен следующей аннотацией:

```
@LogEntry(logger=ИмяРегистратора)
```

то в начале этого метода вводятся байт-коды для приведенного ниже оператора.

```
Logger.getLogger(ИмяРегистратора).entering(ИмяКласса, ИмяМетода);
```

Если, например, снабдить такой аннотацией метод hashCode() из класса Item следующим образом:

```
@LogEntry(logger="global") public int hashCode()
```

то при каждом вызове этого метода будет выводиться приблизительно такое сообщение:

```
May 17, 2016 10:57:59 AM Item hashCode
FINER: ENTRY
```

Чтобы добиться такого результата, необходимо выполнить следующие действия.

1. Загрузить байт-коды в файл класса.
2. Определить местонахождение всех методов.

3. Выполнить для каждого метода проверку на наличие в нем аннотации `LogEntry`.
4. Если имеется такая аннотация, ввести байт-коды для следующих инструкций в начале метода:

```
ldc ИмяРегистратора
invokestatic java/util/logging/Logger.getLogger: (Ljava/lang/String;)Ljava/util/logging/Logger;
ldc ИмяКласса
ldc ИмяМетода
invokevirtual java/util/logging/Logger.entering:
(Ljava/lang/String;Ljava/lang/String;)V
```

Вставка этих байт-кодов может показаться на первый взгляд сложной задачей, но библиотека ASM существенно упрощает ее. Не вдаваясь в подробности анализа и вставки байт-кодов, обратимся к конкретному примеру программы из листинге 8.12. В этой программе редактируется файл классов и вставляется вызов регистратора в начале всех методов, снабженных аннотацией `LogEntry`.

Ниже показано, каким образом инструкции для протоколирования вводятся в исходный файл `Item.java`, представленный в листинге 8.13, где `asm` — каталог, в котором установлена библиотека ASM.

```
javac set/Item.java
javac -classpath .:asm/lib/* bytecodeAnnotations/EntryLogger.java
java -classpath .:asm/lib/* bytecodeAnnotations.EntryLogger set.Item
```

Попробуйте выполнить следующую команду до и после изменения файла класса `Item`:

```
javap -c set.Item
```

В итоге инструкции для протоколирования должны быть вставлены в начале методов `hashCode()`, `equals()` и `compareTo()`:

```
public int hashCode();
Code:
 0: ldc    #85; // String global
 2: invokestatic #80;
   // Method java/util/logging/Logger.getLogger:
   // (Ljava/lang/String;)Ljava/util/logging/Logger;
 5: ldc    #86; // String Item
 7: ldc    #88; // String hashCode
 9: invokevirtual #84;
   // Method java/util/logging/Logger.entering:
   // (Ljava/lang/String;Ljava/lang/String;)V
12: bipush 13
14: aload_0
15: getfield      #2; // Field description:Ljava/lang/String;
18: invokevirtual #15; // Method java/lang/String.hashCode:()I
21: imul
22: bipush 17
24: aload_0
25: getfield      #3; // Field partNumber:I
28: imul
29: iadd
30: ireturn
```

Программа SetTest из листинга 8.14 вставляет объекты типа `Item` в хеш-множество. Если вы запустите ее с измененным файлом класса, то увидите протокольные сообщения, аналогичные приведенным ниже.

```
May 17, 2016 10:57:59 AM Item hashCode
FINER: ENTRY
May 17, 2016 10:57:59 AM Item hashCode
FINER: ENTRY
May 17, 2016 10:57:59 AM Item hashCode
FINER: ENTRY
May 17, 2016 10:57:59 AM Item equals
FINER: ENTRY
[[description=Toaster, partNumber=1729],
 [description=Microwave, partNumber=4104]]
```

Обратите внимание на вызов метода `equals()` при вставке одного и того же элемента дважды. Данный пример демонстрирует эффективность конструирования байт-кодов. Аннотации используются для ввода в прикладную программу директив, а инstrumentальное средство редактирования байт-кодов собирает эти директивы и соответственно видоизменяет инструкции виртуальной машины.

Листинг 8.12. Исходный код из файла bytecodeAnnotations/EntryLogger.java

```
31             String signature, String[] exceptions)
32     {
33         MethodVisitor mv = cv.visitMethod(access, methodName,
34                                         desc, signature, exceptions);
35         return new AdviceAdapter(Opcodes.ASM5, mv, access,
36                                   methodName, desc)
37     }
38     private String loggerName;
39
40     public AnnotationVisitor visitAnnotation(
41                                         String desc, boolean visible)
42     {
43         return new AnnotationVisitor(Opcodes.ASM5)
44         {
45             public void visit(String name, Object value)
46             {
47                 if (desc.equals("LbytecodeAnnotations/LogEntry;"))
48                     && name.equals("logger"))
49                 loggerName = value.toString();
50             }
51         };
52     }
53
54     public void onMethodEnter()
55     {
56         if (loggerName != null)
57         {
58             visitLdcInsn(loggerName);
59             visitMethodInsn(INVOKESTATIC,
60                             "java/util/logging/Logger", "getLogger",
61                             "(Ljava/lang/String;)Ljava/util/logging/Logger;", false);
62             visitLdcInsn(className);
63             visitLdcInsn(methodName);
64             visitMethodInsn(INVOKEVIRTUAL,
65                             "java/util/logging/Logger", "entering",
66                             "(Ljava/lang/String;Ljava/lang/String;)V", false);
67             loggerName = null;
68         }
69     }
70 }
71 }
72 }
73 }
74 /**
75 * Вводит код регистрации записей в указанный класс
76 * @param args Имя файла класса для вставки кода
77 */
78 public static void main(String[] args) throws IOException
79 {
80     if (args.length == 0)
81     {
82         System.out.println("USAGE:
83                         java bytecodeAnnotations.EntryLogger classfile");
84         System.exit(1);
85     }
86     Path path = Paths.get(args[0]);
87     ClassReader reader = new ClassReader(
```

```

88         Files.newInputStream(path));
89     ClassWriter writer = new ClassWriter(
90         ClassWriter.COMPUTE_MAXS | ClassWriter.COMPUTE_FRAMES);
91     EntryLogger entryLogger = new EntryLogger(writer, path
92         .toString().replace(".class", "")
93         .replaceAll("[/\\\\\\]", "."));
94     reader.accept(entryLogger, ClassReader.EXPAND_FRAMES);
95     Files.write(Paths.get(args[0]), writer.toByteArray());
96   }
97 }
```

Листинг 8.13. Исходный код из файла set/Item.java

```

1 package set;
2
3 import java.util.*;
4 import bytecodeAnnotations.*;
5
6 /**
7  * Товар с описанием и номенклатурным номером
8  * @version 1.01 2012-01-26
9  * @author Cay Horstmann
10 */
11 public class Item
12 {
13     private String description;
14     private int partNumber;
15
16     /**
17      * Конструирует объект товара
18      * @param aDescription Описание товара
19      * @param aPartNumber Номенклатурный номер
20     */
21     public Item(String aDescription, int aPartNumber)
22     {
23         description = aDescription;
24         partNumber = aPartNumber;
25     }
26
27     /**
28      * Получить описание данного товара
29      * @return Возвращает описание товара
30     */
31     public String getDescription()
32     {
33         return description;
34     }
35
36     public String toString()
37     {
38         return "[description=" + description + ", partNumber="
39             + partNumber + "]";
40     }
41
42     @LogEntry(logger = "global")
43     public boolean equals(Object otherObject)
```

```
44  {
45      if (this == otherObject) return true;
46      if (otherObject == null) return false;
47      if (getClass() != otherObject.getClass()) return false;
48      Item other = (Item) otherObject;
49      return Objects.equals(description, other.description)
50          && partNumber == other.partNumber;
51  }
52
53  @LogEntry(logger = "global")
54  public int hashCode()
55  {
56      return Objects.hash(description, partNumber);
57  }
58 }
```

Листинг 8.14. Исходный код из файла set/SetTest.java

```
1 package set;
2
3 import java.util.*;
4 import java.util.logging.*;
5
6 /**
7  * @version 1.02 2012-01-26
8  * @author Cay Horstmann
9 */
10 public class SetTest
11 {
12     public static void main(String[] args)
13     {
14         Logger.getLogger("com.horstmann").setLevel(Level.FINEST);
15         Handler handler = new ConsoleHandler();
16         handler.setLevel(Level.FINEST);
17         Logger.getLogger("com.horstmann").addHandler(handler);
18
19         Set<Item> parts = new HashSet<>();
20         parts.add(new Item("Toaster", 1279));
21         parts.add(new Item("Microwave", 4104));
22         parts.add(new Item("Toaster", 1279));
23         System.out.println(parts);
24     }
25 }
```

8.7.2. Видоизменение байт-кодов во время загрузки

В предыдущем разделе было представлено инструментальное средство, способное редактировать файлы классов. Но добавление еще одного такого инструментального средства во время компоновки прикладной программы может оказаться непростым делом. Поэтому существует довольно привлекательный альтернативный способ, который заключается в том, чтобы отложить конструирование байт-кодов до стадии загрузки, на которой загружаются классы.

В прикладном программном интерфейсе API для оснащения инструментальными средствами имеется перехватчик, позволяющий устанавливать преобразователь байт-кодов. Этот преобразователь должен устанавливаться перед вызовом главного метода программы. Чтобы удовлетворить данному требованию, определяется *агент* (т.е. библиотека, загружаемая для контроля над программой). Код этого агента может выполнять в методе `premain()` операции, требующиеся для инициализации.

Чтобы создать агент, необходимо выполнить следующие действия.

1. Реализовать класс с помощью следующего метода:

```
public static void premain(String arg, Instrumentation instr)
```

2. Этот метод вызывается при загрузке агента. Агент может получить единственный аргумент командной строки, передаваемый в качестве параметра `arg`. А параметр `instr` можно использовать для установки различных перехватчиков.
3. Создать файл манифеста, в котором устанавливается атрибут `Premain-Class`, например, следующим образом:

```
Premain-Class: bytecodeAnnotations.EntryLoggingAgent
```

4. Упаковать код агента и манифест в архивный JAR-файл, например, так, как показано ниже.

```
javac -classpath .:asm/lib/*  
       bytecodeAnnotations/EntryLoggingAgent.java  
jar cvfm EntryLoggingAgent.jar  
       bytecodeAnnotations/EntryLoggingAgent.mf \  
       bytecodeAnnotations/Entry*.class
```

Чтобы запустить программу на Java вместе с агентом, необходимо указать следующие параметры в командной строке:

```
java -javaagent:JAR-файл_с_агентом=АргументАгента . . .
```

Например, чтобы запустить программу `SetTest` с агентом протоколирования записей, потребуется выполнить приведенные ниже команды, где аргумент `Item` обозначает имя класса, который агент должен видоизменить.

```
javac set/SetTest.java  
java -javaagent:EntryLoggingAgent.jar=set.Item -classpath  
      .:asm/lib/* set.SetTest
```

В листинге 8.15 представлен исходный код агента, устанавливающего преобразователь файлов классов. Этот преобразователь сначала проверяет, соответствует ли имя класса аргументу агента. Если соответствует, то он использует класс `EntryLogger`, упоминавшийся в предыдущем разделе, для видоизменения байт-кодов. Но видоизмененные байт-коды не сохраняются в файле. Вместо этого преобразователь возвращает их для загрузки в виртуальную машину (рис. 8.3). Иными словами, данный способ позволяет видоизменять байт-коды в динамическом режиме.

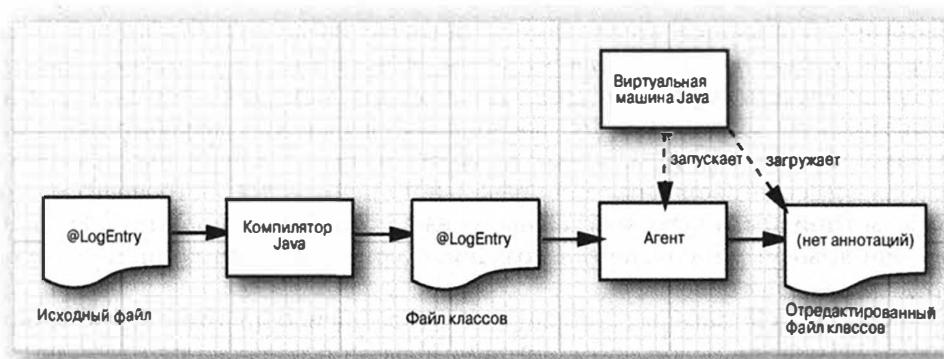


Рис. 8.3. Видоизменение классов во время загрузки

Листинг 8.15. Исходный код из файла `bytecodeAnnotations/EntryLoggingAgent.java`

```

1 package bytecodeAnnotations;
2
3 import java.lang.instrument.*;
4
5 import org.objectweb.asm.*;
6
7 /**
8  * @version 1.10 2016-05-10
9  * @author Cay Horstmann
10 */
11 public class EntryLoggingAgent
12 {
13     public static void premain(final String arg,
14                             Instrumentation instr)
15     {
16         instr.addTransformer((loader, className, cl, pd, data) ->
17         {
18             if (!className.equals(arg)) return null;
19             ClassReader reader = new ClassReader(data);
20             ClassWriter writer = new ClassWriter(
21                 ClassWriter.COMPUTE_MAXS | ClassWriter.COMPUTE_FRAMES);
22             EntryLogger el = new EntryLogger(writer, className);
23             reader.accept(el, ClassReader.EXPAND_FRAMES);
24             return writer.toByteArray();
25         });
26     }
27 }
  
```

Из этой главы вы узнали, как:

- вводить аннотации в исходный код программ на Java;
- создавать свои собственные интерфейсы аннотаций;
- реализовывать инструментальные средства, применяющие аннотации.

В этой главе были также продемонстрированы три методики обработки кода: написание сценариев, компиляция программ на Java и обработка аннотаций. Первые две методики довольно просты. Что же касается создания инструментальных средств обработки аннотаций, то эта методика, безусловно, сложна, и поэтому ею вряд ли рискнут воспользоваться многие разработчики. Следует, однако, иметь в виду, что в этой главе были представлены лишь самые основные сведения, необходимые для правильного понимания внутреннего механизма работы типичных инструментальных средств обработки аннотаций. Но, возможно, они вызовут у вас интерес к созданию своих собственных инструментальных средств для этих целей.

В следующей главе речь пойдет уже о совершенно другом предмете: безопасности. Безопасность всегда была отличительной особенностью платформы Java. В связи с постоянным увеличением степени риска в мире, где мы живем, трудимся и программируем, ясное представление о средствах безопасности Java становится все более актуальным для многих разработчиков.

Безопасность

В этой главе...

- ▶ Загрузчики классов
- ▶ Диспетчеры защиты и полномочия
- ▶ Аутентификация пользователей
- ▶ Цифровые подписи
- ▶ Шифрование

С появлением технологии Java специалисты оценили по достоинству не только хорошо продуманные выразительные средства этого языка, но и механизмы обеспечения безопасности при выполнении аплетов, доставляемых через Интернет. Очевидно, что доставка исполняемых аплетов имеет смысл только тогда, когда получатели уверены, что код не может нанести ущерба работе их компьютеров. Поэтому вопросы безопасности были и остаются основной заботой как разработчиков, так и пользователей технологии Java. Это означает, что, в отличие от других языков и систем, где безопасность обеспечивалась в последнюю очередь, механизмы защиты изначально стали неотъемлемой частью технологии Java.

В технологии Java безопасность обеспечивают следующие три механизма.

- Структурные функциональные возможности языка (например, проверка границ массивов, запрет на преобразования непроверенных типов данных, отсутствие указателей и т.д.).
- Средства контроля доступа, определяющие действия, которые разрешается или запрещается выполнять в коде (например, может ли код получать доступ к файлам, передавать данные по сети и т.д.).
- Механизм цифровой подписи, предоставляющий авторам возможность применять стандартные алгоритмы для аутентификации своих программ, а пользователям — точно определять, кто создал код и изменился ли он с момента его подписания.

В этой главе сначала рассматриваются загрузчики классов, проверяющие файлы классов на предмет целостности при их загрузке в виртуальную машину Java. А затем будет показано, каким образом этот механизм может выявлять в файлах классов признаки злонамеренных действий.

Для обеспечения максимальной безопасности оба механизма загрузки классов (используемый по умолчанию и специальный) должны взаимодействовать с классом диспетчера защиты, определяющим действия, которые разрешено или запрещено выполнять в коде. Поэтому далее в этой главе подробно поясняется, каким образом настраивается безопасность на платформе Java.

И в завершение главы будут рассмотрены различные алгоритмы шифрования, доступные в пакете `java.security` и позволяющие подписывать код и обеспечивать аутентификацию пользователей. Как обычно, основное внимание уделяется только тем вопросам, которые представляют наибольший интерес для разработчиков прикладных программ на Java. Тем, кому требуется более углубленное изучение данной темы, рекомендуем книгу *Inside Java 2 Platform Security: Architecture, API Design, and Implementation, 2nd edition* Ли Гонга, Гэри Эллисона и Мэри Дейджфорда (Li Gong, Gary Ellison, Mary Dageforde; издательство Prentice Hall PTR, 2003 г.).

9.1. Загрузчики классов

Компилятор Java преобразует исходные операторы языка в понятный для виртуальной машины Java байт-код, который сохраняется в файле класса с расширением `.class`. В каждом файле класса содержится код определения и реализации только для одного класса или интерфейса. В последующих разделах будет показано, каким образом виртуальная машина Java загружает файлы классов.

9.1.1. Процесс загрузки классов

Виртуальная машина Java загружает только те файлы классов, которые требуются для выполнения программы в данный момент. Допустим, что выполнение программы начинается с файла `MyProgram.class`. Ниже описаны действия, которые выполняет виртуальная машина Java.

1. В виртуальной машине имеется механизм загрузки файлов классов, например, путем их чтения с диска или загрузки из Интернета. С помощью этого механизма загружается содержимое файла класса `MyProgram`.
2. Если в классе `MyProgram` встречаются поля или объекты, ссылающиеся на классы других типов, то дополнительно загружаются файлы этих классов. (Процесс загрузки всех классов, от которых зависит данный класс, называется *разрешением класса*.)
3. Далее виртуальная машина выполняет метод `main()` из класса `MyProgram`. (Этот метод является статическим, поэтому никаких экземпляров класса `MyProgram` создавать не требуется.)
4. Если для выполнения метода `main()` или вызываемого из него метода требуются дополнительные классы, они загружаются далее из соответствующих файлов.

Но в механизме загрузки классов используется не один, а несколько загрузчиков. Каждой программе на Java сопутствует по меньшей мере три загрузчика классов:

- загрузчик базовых классов;
- загрузчик расширений классов;
- загрузчик системных классов, иногда называемый загрузчиком прикладных классов.

Загрузчик базовых классов загружает базовые или системные классы, как правило, из архивного JAR-файла `rt.jar`. Он является неотъемлемой частью виртуальной машины и обычно реализуется на языке С. Этому загрузчику классов не соответствует ни один из объектов типа `ClassLoader`. Например, при вызове следующего метода возвращается пустое значение `null`:

```
String.class.getClassLoader()
```

Загрузчик расширений классов загружает стандартные расширения из каталога `jre/lib/ext`. Если переместить архивные JAR-файлы в этот каталог, то загрузчик расширений классов найдет в нем нужные классы расширений даже без указания пути к ним. (Некоторые рекомендуют применять этот механизм, чтобы избежать путаницы при указании пути к классам, но такой способ связан с определенными сложностями, поясняемыми в приведенной ниже врезке “Внимание”.)

Загрузчик системных классов загружает системные или прикладные классы, которые размещаются в каталогах и архивных файлах формата JAR/ZIP. Путь к классам должен быть указан в переменной окружения `CLASSPATH` или параметре командной строки `-classpath`.

В версии платформы Java, внедренной компанией Oracle, загрузчики расширений и системных классов реализуются на Java. Оба загрузчика являются экземплярами класса `URLClassLoader`.

Внимание! Если из архива требуется извлечь класс, который, в свою очередь, должен загружать класс, не являющийся системным или расширением, то размещать архивные JAR-файлы в каталоге `jre/lib/ext` нежелательно. Ведь загрузчик расширений классов не пользуется сведениями о пути к классам, устанавливаемыми в переменной окружения `CLASSPATH`. Это обстоятельство следует иметь в виду, принимая решение об использовании каталога `jre/lib/ext` для хранения своих классов.

На заметку! Помимо всех перечисленных выше мест, классы могут загружаться из каталога `jre/lib/endorsed`. Такой механизм может применяться только для замены определенных стандартных библиотек Java (вроде тех, что предназначены для поддержки XML и CORBA) более новыми версиями. Подробнее об этом можно узнать по адресу <http://docs.oracle.com/javase/8/docs/technotes/guides/standards>.

9.1.2. Иерархия загрузчиков классов

Загрузчики классов связаны отношениями “родитель–потомок”. У каждого загрузчика классов, за исключением базовых, имеется свой родительский загрузчик классов. Предполагается, что загрузчик классов дает возможность своему родителю загружать любой нужный класс и загружает его сам только в том случае, если этого не может сделать родитель. Так, если загрузчик системных классов получает

запрос на загрузку системного класса (например, `java.util.ArrayList`), он сначала предлагает загрузить его загрузчику расширенных классов. А загрузчик расширенных классов, в свою очередь, предлагает сделать это загрузчику базовых классов. В итоге загрузчик базовых классов находит и загружает класс из архивного файла `rt.jar`, а все остальные загрузчики классов прекращают дальнейший поиск.

Некоторые программы имеют модульную архитектуру, где определенные части кода упаковываются в дополнительные, но необязательно подключаемые модули. Если подключаемые модули упаковываются в архивные JAR-файлы, то классы этих модулей могут просто загружаться с помощью экземпляра класса `URLClassLoader`, как показано ниже.

```
URL url = new URL("file:///path/to/plugin.jar");
URLClassLoader pluginLoader = new URLClassLoader(new URL[] { url });
Class<?> cl = pluginLoader.loadClass("mypackage.MyClass");
```

В конструкторе класса `URLClassLoader` вообще не указан родитель загрузчика подключаемых модулей типа `pluginLoader`, поэтому в роли его родителя будет выступать загрузчик системных классов. Схематическое представление иерархии загрузчиков классов приведено на рис. 9.1.

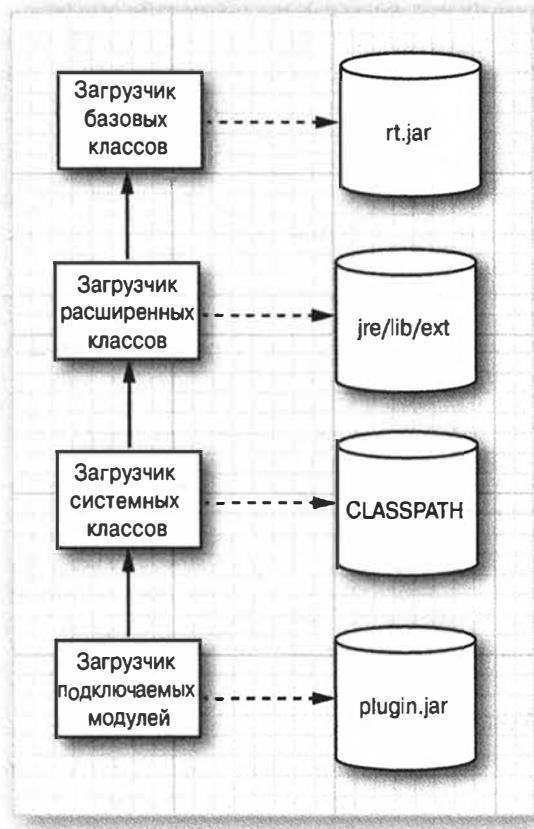


Рис. 9.1. Иерархия загрузчиков классов

Как правило, разработчиков прикладных программ мало интересует иерархия загрузчиков классов. Обычно одни классы загружаются потому, что этого требуют другие классы, и этот процесс является абсолютно прозрачным для разработчика. Но иногда разработчикам прикладных программ все-таки приходится вмешиваться в этот процесс, чтобы указывать нужный загрузчик классов. Рассмотрим следующий пример.

- В коде прикладной программы имеется вспомогательный метод, вызывающий метод `Class.forName(classNameString)`.
- Этот метод вызывается из класса подключаемого модуля.
- Параметр `classNameString` указывает на класс, содержащийся в архивном файле `plugin.jar`.

Автор подключаемого модуля имеет все основания предполагать, что его класс будет загружаться. Но загрузка класса вспомогательного метода выполнялась загрузчиком системных классов, т.е. именно тем загрузчиком, который использовался методом `Class.forName()`. Это означает, что классы, находящиеся в архивном файле `plugin.jar`, недоступны. Подобное явление называется *инверсией загрузчиков классов*.

Для устранения данного препятствия требуется, чтобы вспомогательный метод использовал нужный загрузчик классов. А для этого можно указать нужный загрузчик классов в виде параметра или установить его в качестве загрузчика контекста классов в текущем потоке исполнения. Вторая методика применяется во многих прикладных программных интерфейсах API (например, JAXP и JNDI, упоминавшихся в главах 3 и 5).

В каждом потоке исполнения имеется ссылка на загрузчик классов, называемый загрузчиком контекста классов. Таким загрузчиком для главного потока исполнения является загрузчик системных классов. При создании нового потока исполнения для него назначается загрузчик контекста классов из того потока исполнения, который его создает. Следовательно, если не вмешиваться в этот процесс, то для всех потоков исполнения в качестве загрузчика контекста классов назначается загрузчик системных классов. Тем не менее существует возможность указать любой другой загрузчик классов, сделав следующий вызов:

```
Thread t = Thread.currentThread();
t.setContextClassLoader(loader);
```

Вспомогательный метод может затем извлечь загрузчик контекста классов следующим образом:

```
Thread t = Thread.currentThread();
ClassLoader loader = t.getContextClassLoader();
Class cl = loader.loadClass(className);
```

Теперь остается разрешить следующий вопрос: когда в качестве этого загрузчика контекста классов должен быть назначен загрузчик класса подключаемого модуля? Решение зависит уже от самого разработчика прикладной программы. Обычно лучше сделать так, чтобы загрузчик контекста классов устанавливался при вызове метода из класса подключаемого модуля, который был загружен с помощью другого загрузчика классов. Но, с другой стороны, можно сделать

и так, чтобы загрузчик контекста классов устанавливался в коде, вызывающем вспомогательный метод.



Совет! При написании метода, загружающего класс по имени, вызывающему его коду рекомендуется предоставлять возможность выбирать между передачей явно задаваемого загрузчика классов и применением загрузчика контекста классов. А просто использовать загрузчик класса этого метода нежелательно.

9.1.3. Применение загрузчиков классов в качестве пространств имен

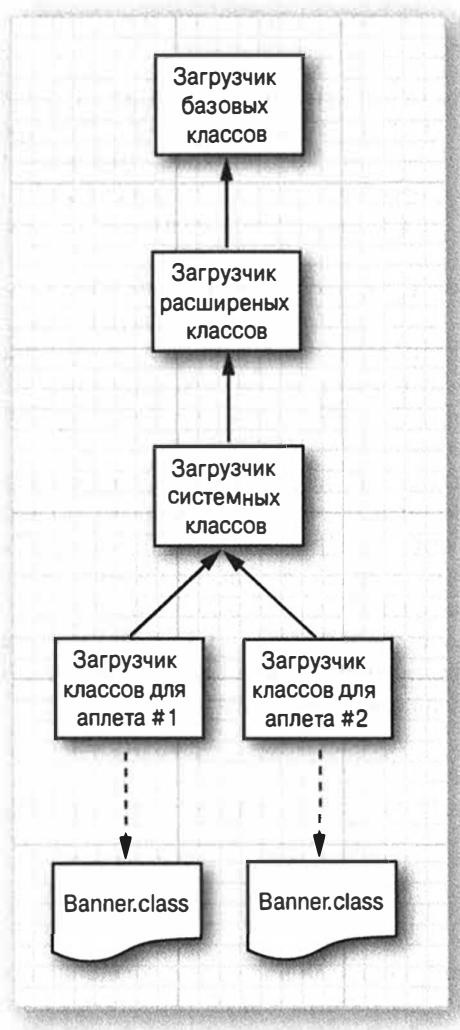


Рис. 9.2. Два загрузчика классов загружают разные классы аплетов с одинаковыми именами

Всякому программирующему на Java известно, что для устранения конфликтов имен применяются имена пакетов. Например, в стандартной библиотеке Java имеются два класса под именем Date, но полностью они именуются как java.util.Date и java.sql.Date. Короткие имена употребляются для удобства программирования, но требуют включения соответствующих операторов import в исходный код. А в выполняющейся программе все имена классов обязательно содержат имена своих пакетов.

Но, как ни странно, в пределах одной и той же виртуальной машины могут существовать два разных класса, имеющих одинаковое имя класса и пакета. Дело в том, что класс определяется по его полному имени и загрузчику класса. Такая технология очень удобна при загрузке кода из нескольких источников. Например, в веб-браузере для загрузки каждой веб-страницы используются отдельные экземпляры загрузчика классов аплетов. Это позволяет виртуальной машине различать классы разных веб-страниц независимо от их имен. Допустим, что веб-страница содержит два аплета, предоставляемых двумя разными рекламодателями, и у каждого из этих аплетов имеется класс Banner. Поскольку каждый аплет загружается отдельным загрузчиком классов, эти классы являются совершенно отдельными и не конфликтуют друг с другом (рис. 9.2).



На заметку! Существуют и другие способы использования данной технологии, например, "горячее развертывание" сервлетов и компонентов Enterprise JavaBeans. Подробнее об этом можно узнать по адресу <http://zeroturnaround.com/labs/rjc301>.

9.1.4. Создание собственного загрузчика классов

Для специальных целей можно создавать и свои собственные загрузчики классов. Такой подход позволяет выполнять перед передачей байт-кода виртуальной машине какие-нибудь специальные проверки. Например, можно создать загрузчик классов, способный запрещать загрузку класса, который не был помечен как "оплаченный". Чтобы создать собственный загрузчик классов, достаточно расширить класс `ClassLoader` и переопределить метод `findClass(String className)`.

Метод `loadClass()` из суперкласса `ClassLoader` берет на себя все хлопоты по делегированию функций загрузки родительскому загрузчику классов и вызывает метод `findClass()` только в том случае, если класс еще не был загружен и если родительскому загрузчику классов не удалось его загрузить. Поэтому при самостоятельной реализации метода `findClass()` необходимо сделать так, чтобы он выполнял следующее.

1. Загружал байт-код класса из локальной файловой системы или какого-нибудь другого источника.
2. Вызывал метод `defineClass()` из суперкласса `ClassLoader` для представления байт-кода виртуальной машине.

В примере программы из листинга 9.1 демонстрируется реализация загрузчика классов, способного загружать файлы зашифрованных классов. Эта программа сначала запрашивает у пользователя имя первого загружаемого класса (т.е. класса, содержащего метод `main()`) и ключ расшифровки. Затем она использует специальный загрузчик классов для загрузки указанного класса и вызова метода `main()`. Этот специальный загрузчик расшифровывает указанный класс и все несистемные классы, на которые он ссылается, после чего программа наконец-то вызывает метод `main()` из загруженного класса (рис. 9.3). Ради простоты для шифрования файлов классов в рассматриваемом здесь примере используется древний шифр Юлия Цезаря, пренебрегая более чем 2000-летним опытом в области криптографии.



На заметку! В прекрасной книге *The Codebreakers* Дэвида Кана [David Kahn; издательство Macmillan, 1967 г.; в русском переводе она вышла под названием *Взломщики кодов* в издательстве "Центрполиграф", 2000 г.] говорится, что Юлий Цезарь кодировал 24 буквы латинского алфавита, сдвигая их на 3 буквы, чем повергал в недоумение всех своих противников. На момент написания первоначального варианта этой главы правительство США ограничивало экспорт наиболее сложных методов шифрования. Поэтому использование довольно простого метода шифрования Юлия Цезаря в рассматриваемом здесь примере никак не нарушит экспортные ограничения.

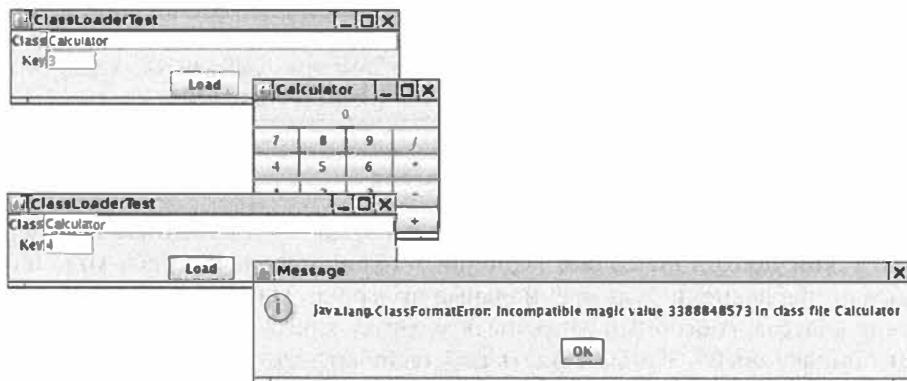


Рис. 9.3. Программа ClassLoaderTest

В рассматриваемом здесь варианте тайнописи Юлия Цезаря в качестве ключа используется число от 1 до 255. Для шифрования байт суммируется с этим ключом по модулю 256. Алгоритм шифрования реализован в программе из файла Caesar.java, исходный код которой приведен в листинге 9.2. Чтобы не поставить в тупик стандартный загрузчик классов, для зашифрованных файлов классов используется расширение .caesar.

При расшифровке загрузчик классов вычитает значение ключа из каждого байта. В прилагаемом к данной книге коде предлагаются четыре файла классов, зашифрованных с помощью традиционного значения 3 ключа шифрования. Эти классы нельзя загрузить и запустить на выполнение в стандартной виртуальной машине Java. Для этого понадобится специальный загрузчик классов, реализованный в рассматриваемой здесь программе ClassLoaderTest.

Зашифрованные файлы классов имеют самое разное практическое применение (если, конечно, используется более сложный шифр, чем тайнопись Юлия Цезаря). Без ключа шифрования эти файлы классов бесполезны, поскольку они не могут выполняться в стандартной виртуальной машине Java, а восстановить их исходный код нелегко.

Это означает, что для аутентификации пользователя класса или для проверки того, что программа оплачена до запуска, можно воспользоваться специальным загрузчиком классов. Естественно, шифрование является только одним из возможных применений специального загрузчика классов. Для решения других задач, например, сохранения файлов классов в базе данных, можно создавать и применять иные разновидности загрузчиков классов.

Листинг 9.1. Исходный код из файла classLoader/ClassLoaderTest.java

```

1 package classLoader;
2
3 import java.io.*;
4 import java.lang.reflect.*;
5 import java.nio.file.*;
6 import java.awt.*;
7 import java.awt.event.*;
8 import javax.swing.*;
```

```
9
10 /**
11  * В этой программе демонстрируется специальный загрузчик
12  * классов, расшифровывающий файлы классов
13  * @version 1.24 2016-05-10
14  * @author Cay Horstmann
15 */
16 public class ClassLoaderTest
17 {
18     public static void main(String[] args)
19     {
20         EventQueue.invokeLater(() ->
21             {
22                 JFrame frame = new ClassLoaderFrame();
23                 frame.setTitle("ClassLoaderTest");
24                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25                 frame.setVisible(true);
26             });
27     }
28 }
29
30 /**
31  * Этот фрейм содержит два текстовых поля для ввода имени
32  * загружаемого класса и ключ расшифровки
33 */
34 class ClassLoaderFrame extends JFrame
35 {
36     private JTextField keyField = new JTextField("3", 4);
37     private JTextField nameField = new JTextField("Calculator", 30);
38     private static final int DEFAULT_WIDTH = 300;
39     private static final int DEFAULT_HEIGHT = 200;
40
41     public ClassLoaderFrame()
42     {
43         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
44         setLayout(new GridBagLayout());
45         add(new JLabel("Class"), new GBC(0, 0).setAnchor(GBC.EAST));
46         add(nameField, new GBC(1, 0).setWeight(100, 0)
47             .setAnchor(GBC.WEST));
48         add(new JLabel("Key"), new GBC(0, 1).setAnchor(GBC.EAST));
49         add(keyField, new GBC(1, 1).setWeight(100, 0)
50             .setAnchor(GBC.WEST));
51         JButton loadButton = new JButton("Load");
52         add(loadButton, new GBC(0, 2, 2, 1));
53         loadButton.addActionListener(event ->
54             runClass(nameField.getText(), keyField.getText()));
55         pack();
56     }
57
58 /**
59  * Выполняет основной метод указанного класса
60  * @param name Имя класса
61  * @param key Ключ расшифровки файлов классов
62  */
63     public void runClass(String name, String key)
64     {
65         try
66         {
```

```
67     ClassLoader loader = new CryptoClassLoader(
68         Integer.parseInt(key));
69     Class<?> c = loader.loadClass(name);
70     Method m = c.getMethod("main", String[].class);
71     m.invoke(null, (Object) new String[] {});
72 }
73 catch (Throwable e)
74 {
75     JOptionPane.showMessageDialog(this, e);
76 }
77 }
78
79 }
80
81 /**
82 * Этот загрузчик классов загружает их из зашифрованных файлов
83 */
84 class CryptoClassLoader extends ClassLoader
85 {
86     private int key;
87
88     /**
89      * Конструирует загрузчик зашифрованных классов
90      * @param k Ключ расшифровки
91      */
92     public CryptoClassLoader(int k)
93     {
94         key = k;
95     }
96
97     protected Class<?> findClass(String name)
98         throws ClassNotFoundException
99     {
100         try
101         {
102             byte[] classBytes = null;
103             classBytes = loadClassBytes(name);
104             Class<?> cl = defineClass(name, classBytes,
105                                         0, classBytes.length);
106             if (cl == null) throw new ClassNotFoundException(name);
107             return cl;
108         }
109         catch (IOException e)
110         {
111             throw new ClassNotFoundException(name);
112         }
113     }
114
115     /**
116      * Загружает и расшифровывает байты из файла класса
117      * @param name Имя класса
118      * @return Возвращает массив с байтами из файла класса
119      */
120     private byte[] loadClassBytes(String name) throws IOException
121     {
122         String cname = name.replace('.', '/') + ".caesar";
123         byte[] bytes = Files.readAllBytes(Paths.get(cname));
124         for (int i = 0; i < bytes.length; i++)
```

```
125     bytes[i] = (byte) (bytes[i] - key);
126     return bytes;
127 }
128 }
```

Листинг 9.2. Исходный код из файла classLoader/Caesar.java

```
1 package classLoader;
2
3 import java.io.*;
4
5 /**
6  * Шифрует файл, используя тайнопись Юлия Цезаря
7  * @version 1.01 2012-06-10
8  * @author Cay Horstmann
9 */
10 public class Caesar
11 {
12     public static void main(String[] args) throws Exception
13     {
14         if (args.length != 3)
15         {
16             System.out.println("USAGE:
17                             java classLoader.Caesar in out key");
18             return;
19         }
20
21         try(FileInputStream in = new FileInputStream(args[0]);
22             FileOutputStream out = new FileOutputStream(args[1]))
23         {
24             int key = Integer.parseInt(args[2]);
25             int ch;
26             while ((ch = in.read()) != -1)
27             {
28                 byte c = (byte) (ch + key);
29                 out.write(c);
30             }
31         }
32     }
33 }
```

java.lang.Class 1.0

- **ClassLoader getClassLoader()**
Получает загрузчик для загрузки данного класса.

java.lang.ClassLoader 1.0

- **ClassLoader getParent() 1.2**
Получает родительский загрузчик классов или пустое значение **null**, если это загрузчик базовых классов.

java.lang.ClassLoader 1.0 (окончание)**• static ClassLoader getSystemClassLoader() 1.2**

Получает загрузчик системных классов, т.е. тот, который применялся для загрузки первого прикладного класса.

• protected Class findClass(String name) 1.2

Должен быть переопределен в загрузчике классов для поиска байт-кода класса и его представления виртуальной машине благодаря вызову метода **defineClass()**. Для отделения имени пакета от имени класса следует использовать точку и не указывать суффикс **.class**.

• Class defineClass(String name, byte[] byteCodeData, int offset, int length)

Передает виртуальной машине новый класс, предоставляя байт-код в определенном диапазоне данных.

java.net.URLClassLoader 1.2**• URLClassLoader(URL[] urls)****• URLClassLoader(URL[] urls, ClassLoader parent)**

Создает загрузчик классов по указанному URL. Если URL оканчивается знаком **/**, то подразумевается каталог, а иначе — архивный JAR-файл.

java.lang.Thread 1.0**• ClassLoader getContextClassLoader() 1.2**

Получает загрузчик классов, обозначенный создателем данного потока исполнения как наиболее приемлемый для использования в этом потоке.

• void setContextClassLoader(ClassLoader loader) 1.2

Устанавливает загрузчик классов для кода в данном потоке исполнения. Если при запуске потока на выполнение загрузчик контекста классов не задан явно, то используется родительский загрузчик контекста классов.

9.1.5. Верификация байт-кода

Когда загрузчик классов представляет виртуальной машине байт-код класса, этот код сначала обследуется *верификатором*. Верификатор проверяет все классы за исключением системных и определяет те команды, которые могут нанести ущерб виртуальной машине.

Ниже приведены некоторые виды проверок, выполняемых верификатором.

- Инициализация переменных перед их использованием.
- Согласование типов ссылок при вызове метода.
- Соблюдение правил доступа к закрытым данным и методам.
- Доступ к локальным переменным в стеке во время выполнения.
- Отсутствие переполнения стека.

При невыполнении какой-нибудь из этих проверок класс считается поврежденным и загружаться не будет.



На заметку! Теорема Гёделя утверждает, что невозможно разработать алгоритмы, способные обрабатывать программные файлы и принимать решение, обладают ли программы ввода данных конкретным свойством (например, независимостью от переполнения стека). В таком случае возникает вопрос: каким образом верификатор может определить, что в файле класса не соблюдается соответствие типов, используются неинициализированные переменные и переполняется стек? Не противоречит ли это законам логики? Никакого противоречия нет, потому что верификатор не является алгоритмом принятия решений в смысле теоремы Гёделя. Если верификатор принимает программу, значит, она действительно безопасна. Но он может и отклонять команды виртуальной машины, даже если те выглядят вполне безопасными. (Вам, вероятно, уже доводилось сталкиваться с подобным затруднением и инициализировать переменную каким-нибудь фиктивным значением из-за того, что компилятор не в состоянии гарантировать еенюю инициализацию.)

Такая строгая верификация требуется по соображениям безопасности. Случайные ошибки, связанные с использованием неинициализированных переменных, могут легко нанести значительный ущерб программе, если вовремя не определить их. Еще важнее то обстоятельство, что при повсеместном доступе к Интернету необходимо обеспечить надежную защиту от злонамеренных попыток недобросовестных программистов, стремящихся проникнуть в чужую систему или нанести ей ущерб. Например, видоизменив значения в стеке выполняемой программы или в закрытых полях системных объектов, любая программа способна нарушить систему защиты веб-браузера.

Зачем же создавать специальный верификатор для проверки всех этих условий? Ведь компилятор все равно не позволит сгенерировать файл класса, в котором предполагается использовать неинициализированную переменную или область закрытых данных, доступную из другого класса. В самом деле, файл класса, сформированный компилятором Java, всегда проходит верификацию. Но используемый в этих файлах формат байт-кода хорошо документирован, и всякий, имеющий опыт программирования на ассемблере и работы в шестнадцатеричном редакторе, может без труда создать вручную файл класса с достоверными, но потенциально опасными командами для виртуальной машины Java. Таким образом, верификатор блокирует злонамеренно измененные файлы классов, а не только проверяет файлы, созданные компилятором.

В листинге 9.3 приведен пример программы `VerifierTest`, демонстрирующей изменение файла класса вручную. В этой простой программе вызывается метод `fun()` и выводится результат его выполнения. Она может быть запущена из командной строки или в виде аплекса. Метод `fun()` выполняет сложение чисел 1 и 2, как показано ниже.

```
static int fun()
{
    int m;
    int n;
    m = 1;
    n = 2;
    int r = m + n;
    return r;
}
```

Листинг 9.3. Исходный код из файла verifier/VerifierTest.java

```

1 package verifier;
2
3 import java.applet.*;
4 import java.awt.*;
5 /**
6  * В этой прикладной программе демонстрируется верификатор
7  * байт-кода виртуальной машины Java. Если воспользоваться
8  * шестнадцатеричным редактором для видоизменения файла класса,
9  * то виртуальная машина Java должна обнаружить злонамеренное
10 * искажение содержимого файла класса
11 * @version 1.00 1997-09-10
12 * @author Cay Horstmann
13 */
14 public class VerifierTest extends Applet
15 {
16     public static void main(String[] args)
17     {
18         System.out.println("1 + 2 == " + fun());
19     }
20
21 /**
22  * функция, вычисляющая сумму чисел 1 + 2
23  * @return Возвращает сумму 3, если код не нарушен
24 */
25 public static int fun()
26 {
27     int m;
28     int n;
29     m = 1;
30     n = 2;
31     // воспользоваться шестнадцатеричным редактором, чтобы
32     // изменить значение переменной m на 2 в файле класса
33     int r = m + n;
34     return r;
35 }
36
37 public void paint(Graphics g)
38 {
39     g.drawString("1 + 2 == " + fun(), 20, 20);
40 }
41 }
```

В качестве эксперимента попробуйте скомпилировать приведенный ниже видоизмененный вариант метода `fun()`.

```

static int fun()
{
    int m = 1;
    int n;
    m = 1;
    m = 2;
    int r = m + n;
    return r;
}
```

В данном случае переменная `n` не инициализирована и может принимать произвольное значение. Естественно, что компилятор обнаружит эту ошибку

и откажется компилировать программу. Для создания недопустимого файла класса запустите утилиту `javap`, чтобы выяснить, каким образом компилятор транслирует метод `fun()`:

```
javap -c verifier.VerifierTest
```

Эта утилита показывает байт-код из файла класса в мнемоническом виде:

```
Method int fun()
  0  iconst_1
  1  istore_0
  2  iconst_2
  3  istore_1
  4  iload_0
  5  iload_1
  6  iadd
  7  istore_2
  8  iload_2
  9  ireturn
```

Чтобы изменить команду 3, т.е. заменить `istore_1` на `istore_0`, воспользуйтесь шестнадцатеричным редактором. Таким образом, локальная переменная 0 (т.е. `m`) будет инициализирована дважды, а локальная переменная 1 (т.е. `n`) вообще не будет инициализирована. Чтобы сделать это, необходимо знать шестнадцатеричные значения команд. Эти значения приводятся ниже и взяты из книги *The Java Virtual Machine Specification* Тима Линдхольма и Фрэнка Йеллина (Tim Lindholm & Frank Yellin; издательство Prentice Hall PTR, 1999 г.).

```
0  iconst_1 04
1  istore_0 3B
2  iconst_2 05
3  istore_1 3C
4  iload_0 1A
5  iload_1 1B
6  iadd   60
7  istore_2 3D
8  iload_2 1C
9  ireturn AC
```

Отредактируйте любое из этих значений можно в шестнадцатеричном редакторе. На рис. 9.4 приведено рабочее окно шестнадцатеричного редактора Gnome с загруженным файлом `VerifierTest.class`, в котором светло-серым выделены байт-коды метода `fun()`.

Замените шестнадцатеричное значение 3C на 3B, сохраните файл, а затем попытайтесь запустить программу `VerifierTest` на выполнение. В итоге вы получите следующее сообщение об ошибке:

```
Exception in thread "main" java.lang.VerifyError:
(class: VerifierTest, method:fun signature: ()I)
Accessing value from uninitialized register 1
```

Таким образом, виртуальная машина Java обнаружила недопустимое видоизменение байт-кода. А теперь запустите программу с параметром `-noverify` (или `-Xverify:none`), т.е. без верификации:

```
java -noverify verifier.VerifierTest
```

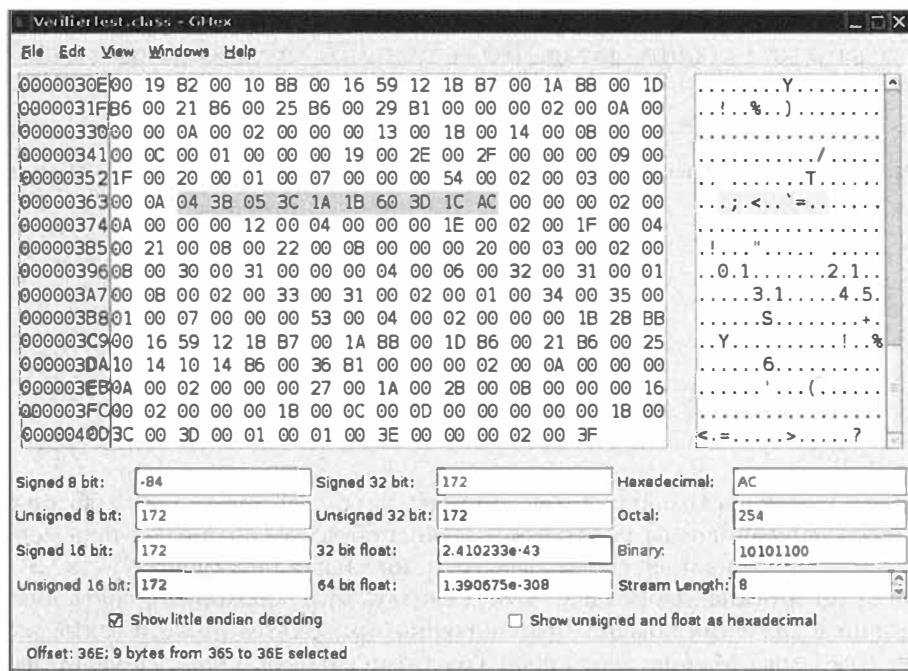


Рис. 9.4. Видоизменение байт-кода в шестнадцатеричном редакторе

Метод `fun()` возвращает случайную сумму, получаемую в результате сложения числа 2 с произвольным значением, хранящимся в переменной `n`, которая не была инициализирована. Ниже приведен типичный результат выполнения рассматриваемой здесь программы без верификации байт-кода.

`1 + 2 == 15102330`

Чтобы продемонстрировать, каким образом верификация производится в веб-браузере, в рассматриваемой здесь программе предусмотрена возможность ее запуска в виде аплета. Загрузите эту программу как аплет в веб-браузер по следующему URL:

`file:///C:/CoreJavaBook/v2ch9/verifier/VerifierTest.html`

Как показано на рис. 9.5, в результате выполнения данной программы появится сообщение об ошибке, обнаруженной во время верификации байт-кода.

9.2. Диспетчеры защиты и полномочия

После загрузки класса в виртуальную машину Java и проверки верификатором в действие вступает второй механизм обеспечения безопасности на платформе Java: диспетчер защиты. Этой теме посвящены последующие разделы.

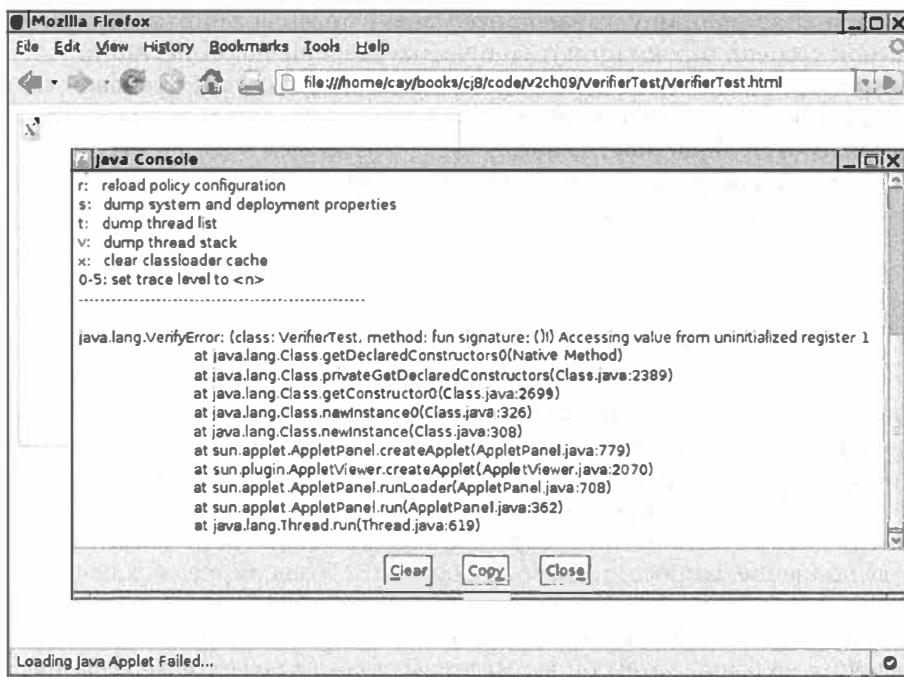


Рис. 9.5. Попытка загрузить преднамеренно видоизмененный файл класса приводит к ошибке верификации байт-кода

9.2.1. Проверка полномочий

Диспетчер защиты проверяет, разрешено ли коду выполнять ту или иную операцию. Ниже перечислены операции, подпадающие под контроль диспетчера защиты. Существует немало других проверок, выполняемых диспетчером защиты в библиотеке Java.

- Создание нового загрузчика классов.
- Выход из виртуальной машины.
- Получение доступа к члену другого класса с помощью рефлексии.
- Получение доступа к файлу.
- Установление соединения через сокет.
- Запуск задания на печать.
- Получение доступа к системному буферу обмена.
- Получение доступа к очереди событий в AWT.
- Обращение к окну верхнего уровня.

Поведение по умолчанию при запуске прикладных программ на Java не предусматривает никакой установки диспетчера защиты, поэтому все перечисленные выше операции оказываются разрешенными. Тем не менее программа

просмотра аплетов принудительно применяет правила защиты, которые в значительной степени ограничивают количество разрешенных операций.

Например, аплетам не разрешается завершать работу виртуальной машины Java. При попытке аплета вызвать метод `exit()` выдается исключение системы безопасности. В частности, метод `exit()` из класса `Runtime` вызывает метод `checkExit()` диспетчера защиты. Ниже приведен исходный код метода `exit()`.

```
public void exit(int status)
{
    SecurityManager security = System.getSecurityManager();
    if (security != null)
        security.checkExit(status);
    exitInternal(status);
}
```

Далее диспетчер защиты проверяет, откуда поступил запрос на выход (т.е. на завершение работы виртуальной машины): из браузера или из отдельного аплета. Если диспетчер защиты дает разрешение на выполнение этого запроса, метод `checkExit()` просто возвращает управление, и далее процесс обработки продолжается обычным образом. Но если диспетчер защиты не запрещает выполнение запроса, то метод `checkExit()` генерирует исключение типа `SecurityException`.

Метод `exit()` продолжает выполняться только в том случае, если никакого исключения не было. Затем он вызывает закрытый платформенно-ориентированный метод `exitInternal()`, который и завершает работу виртуальной машины Java. Другого способа для завершения работы виртуальной машины Java не существует, а поскольку метод `exitInternal()` закрытый, то он не может вызываться из какого-нибудь другого класса. Следовательно, всякий код, делающий попытку выйти из виртуальной машины Java, должен вызывать метод `exit()` и проходить проверку защиты, выполняемую методом `checkExit()`, но без генерирования соответствующего исключения.

Очевидно, что целостность правил защиты зависит от тщательности программирования. Поставщики системных служб в стандартной библиотеке должны всегда обращаться к диспетчеру защиты перед попытками выполнить любые серьезные операции.

Диспетчер защиты на платформе Java позволяет как программистам, так и системным администраторам организовать тщательно продуманное управление отдельными полномочиями. Более подробно об этом речь пойдет в следующем разделе, где сначала приводится краткий обзор модели защиты, реализованной на платформе Java 2, а затем показывается, как организуется управление правами доступа с помощью файлов правил защиты, и объясняется, как определять свои собственные виды прав доступа.



На заметку! Допускается реализовывать и устанавливать свой собственный диспетчер защиты, но это лучше делать только тем, у кого имеется достаточный опыт работы в области компьютерной безопасности. Намного безопаснее просто настраивать нужным образом стандартный механизм диспетчера защиты.

9.2.2. Организация защиты на платформе Java

Модель защиты в JDK версии 1.0 была очень проста: локальные классы имели все полномочия, а возможности удаленных классов были ограничены так называемой "песочницей", т.е. диспетчер защиты аплета отказывал в любом доступе к локальным ресурсам. Подобно тому, как ребенку разрешается играть в песочнице, строго ограничивая его действия ее пределами, удаленные классы имели возможность лишь выводить сведения на экран и взаимодействовать с пользователем. В версии JDK 1.1 эта модель была немного видоизменена, т.е. доверенные удаленные объекты имели те же права доступа, что и локальные классы. Тем не менее обе версии модели защиты были организованы по принципу "все или ничего", т.е. программы получали полный доступ к ресурсам, или же их действие ограничивалось пределами "песочницы".

Начиная с версии Java SE 1.2, на платформе Java предоставляется намного более гибкий механизм защиты. *Правила защиты* преобразуют источники кода в наборы полномочий, как показано на рис. 9.6.

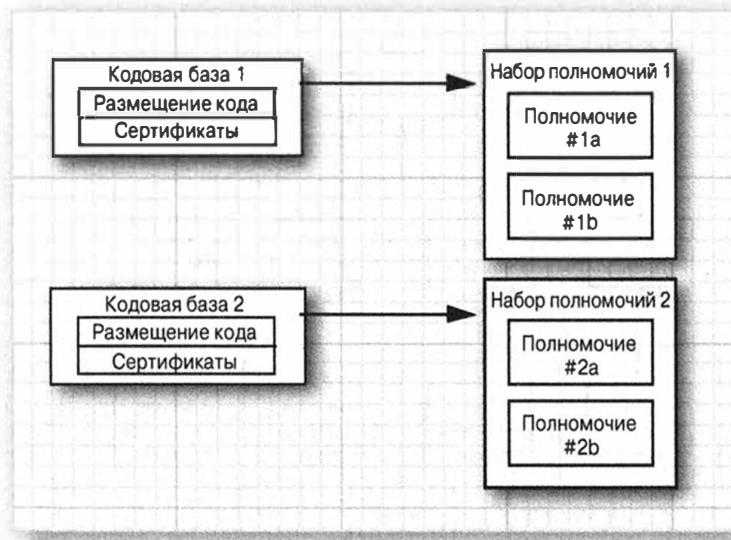


Рис. 9.6. Правила защиты

Каждый источник кода характеризуется кодовой базой и набором сертификатов. Кодовая база указывает место происхождения кода. Например, кодовой базой кода удаленного аплета является URL типа HTTP, откуда этот аплет загружен, а кодовой базой кода в архивном JAR-файле – URL, по которому загружен этот файл. Сертификаты, если таковые имеются, предоставляются соответствующей организацией и служат гарантией того, что данный код не является подделкой. Подробнее о сертификатах речь пойдет далее в этой главе.

Под *полномочиями* (иначе — правами доступа или привилегиями) подразумеваются любые свойства, которые проверяются диспетчером защиты. На платформе Java поддерживается целый ряд представляющих полномочия классов, каждый из которых инкапсулирует подробности конкретных полномочий. В качестве примера ниже приведено получение экземпляра класса `FilePermission`, дающего разрешение на чтение и запись любого файла в каталоге `/tmp`.

```
FilePermission p = new FilePermission("/tmp/*", "read,write");
```

Но еще важнее, что в стандартной реализации класс `Policy` считывает полномочия из файла *прав доступа*. В этом файле те же самые полномочия на чтение и запись, что и выше, выражаются следующим образом:

```
permission java.io.FilePermission "/tmp*", "read,write";
```

Более подробно файлы прав доступа рассматриваются в следующем разделе. На рис. 9.7 представлена иерархия классов полномочий, которые предоставлялись в версии Java SE 1.2. А в последующих версиях Java было внедрено немало других классов полномочий.

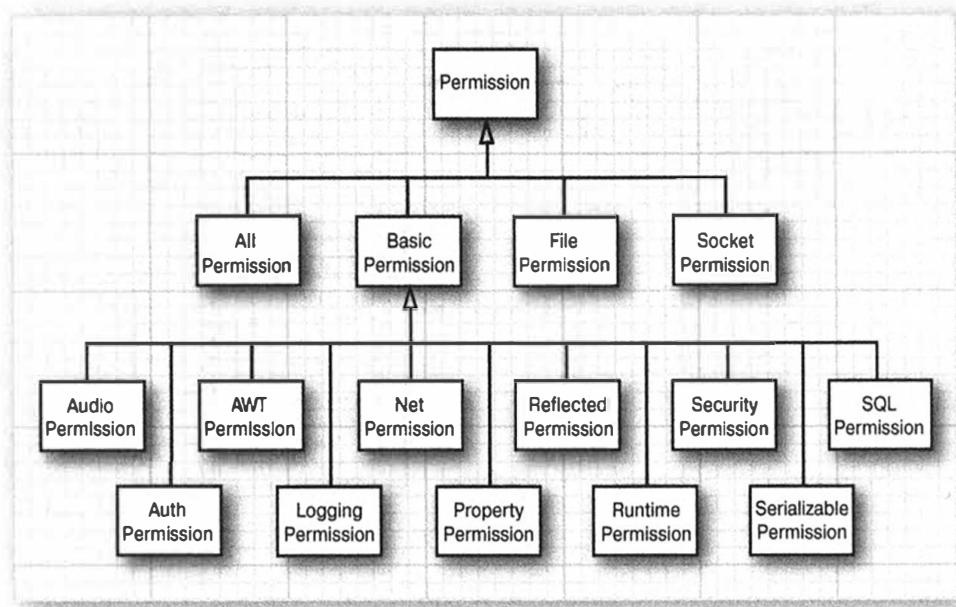


Рис. 9.7. Часть существующей иерархии классов полномочий

Как было показано в предыдущем разделе, в классе `SecurityManager` имеются методы проверки защиты типа `checkExit()`, предназначенные только для удобства программирования и обратной совместимости. Все они отображаются в стандартные проверки полномочий. В качестве примера ниже приведен исходный код метода `checkExit()`.

```
public void checkExit()
{
    checkPermission(new RuntimePermission("exitVM"));
}
```

У каждого класса имеется *домен защиты*, т.е. объект, который инкапсулирует как источник кода, так и набор прав доступа этого класса. При необходимости проверить то или иное полномочие диспетчер защиты типа `SecurityManager` сначала выясняет, к каким классам относятся все методы, находящиеся в настоящий момент в стеке вызовов. Затем он получает доступ к доменам защиты всех этих классов и выясняет, допускает ли их набор полномочий выполнение проверяемой в данный момент операции. Если все домены дают разрешение на выполнение данной операции, то проверка завершается успешно, в противном случае генерируется исключение типа `SecurityException`.

А почему разрешение на выполнение операции должны давать все методы в стеке вызовов? Для ответа на этот вопрос обратимся к конкретному примеру. Допустим, что в методе `init()` из аплета требуется открыть файл с помощью следующего вызова:

```
Reader in = new FileReader(name);
```

Конструктор класса `FileReader` вызывает конструктор класса `InputStream`, который, в свою очередь, обращается к методу `checkRead()` диспетчера защиты, а тот вызывает метод `checkPermission()` с объектом, получаемым из конструктора `FilePermission(name, "read")`. Внешний вид стека вызовов в этом случае приведен в табл. 9.1.

Таблица 9.1. Стек вызовов во время проверки полномочий

Класс	Метод	Источник кода	Полномочия
<code>SecurityManager</code>	<code>checkPermission</code>	null	<code>AllPermission</code>
<code>SecurityManager</code>	<code>checkRead</code>	null	<code>AllPermission</code>
<code>InputStream</code>	<code>Constructor</code>	null	<code>AllPermission</code>
<code>FileReader</code>	<code>Constructor</code>	null	<code>AllPermission</code>
<code>Applet</code>	<code>init</code>	Источник кода аплета	Полномочия аплета

Классы `InputStream` и `SecurityManager` являются системными, их источник кода — пустым значением `null`, а набор полномочий — экземпляром класса `AllPermission`, который разрешает выполнять все операции. Очевидно, что полномочия только этих классов не могут определить исход проверки. Помимо них, метод `checkPermission()` должен принять во внимание и ограниченные полномочия класса аплета. Благодаря проверке всего стека вызовов механизм защиты исключает вероятность выполнения одним классом важных операций от имени какого-нибудь другого класса.



На заметку! Приведенное выше краткое описание проверки полномочий дает лишь самое общее представление о данном процессе. Многие технические детали в этом описании были опущены. Учитывая тот факт, что в вопросах безопасности вся суть скрывается именно в деталях, для ознакомления с ними рекомендуется книга *Securing Java: Getting Down to Business with Mobile Code* Гэри Макгроу и Эда Фельтена [Gary McGraw & Ed Felten; издательство Wiley, 1999 г.]. Электронная версия книги оперативно доступна по адресу www.securingjava.com.

java.lang.SecurityManager 1.0

- **void checkPermission(Permission p) 1.2**

Проверяет, предоставляет ли диспетчер защиты указанные полномочия. Если не предоставляет, то генерируется исключение типа **SecurityException**.

java.lang.Class 1.0

- **ProtectionDomain getProtectionDomain() 1.2**

Получает доступ к представляющему данный класс домену защиты или возвращает пустое значение **null**, если этот класс загружен без домена защиты.

a.security.ProtectionDomain 1.2

- **ProtectionDomain(CodeSource source, PermissionCollection permissions)**
Создает домен защиты, исходя из указанного источника кода и прав доступа.
- **CodeSource getCodeSource()**
Получает источник кода из данного домена защиты.
- **boolean implies(Permission p)**
Возвращает логическое значение **true**, если указанные полномочия разрешены в данном домене защиты.

a.security.CodeSource 1.2

- **Certificate[] getCertificates()**
Получает цепочку сертификатов для подписей файлов классов, связанных с данным источником кода.
- **URL getLocation()**
Получает кодовую базу файлов классов, связанных с данным источником кода.

9.2.3. Файлы правил защиты

Диспетчер правил защиты считывает содержимое файлов правил защиты, т.е. инструкции для преобразования источников кода в полномочия. Ниже приведен пример типичного файла правил защиты. Этот файл предоставляет полномочия на чтение и запись файлов в каталог /tmp любому коду, загруженному по адресу <http://www.horstmann.com/classes>.

```
grant codeBase "http://www.horstmann.com/classes"
{
    permission java.io.FilePermission "/tmp/*", "read,write";
};
```

Устанавливаться файлы правил защиты могут только в стандартных местах. По умолчанию такими местами являются:

- файл `java.policy` в основном каталоге платформы Java;
- файл `.java.policy` в рабочем каталоге пользователя (обратите внимание на точку, стоящую в начале его имени).

 **На заметку!** Стандартные пути к этим файлам можно изменить в конфигурационном файле `java.security`, находящемся в каталоге `jre/lib/security`. По умолчанию пути к этим файлам указываются в конфигурационном файле и выглядят следующим образом:

```
policy.url.1=file:${java.home}/lib/security/java.policy  
policy.url.2=file:${user.home}/.java.policy
```

Системный администратор может редактировать файл `java.security` и указывать для правил защиты URL, находящиеся на каком-нибудь другом сервере, исключая всякую возможность для пользователей вносить изменения в эти правила. В файле правил защиты можно указать сколько угодно URL (по порядку следования номеров). Полномочия во всех подобных файлах все равно объединяются.

Если требуется, чтобы правила защиты хранились вне файловой системы, можно сначала реализовать подкласс, производный от класса `Policy` и собирающий сведения обо всех полномочиях, а затем изменить строку в конфигурационном файле `java.security` следующим образом:

```
policy.provider=sun.security.provider.PolicyFile
```

Во время тестирования не имеет смысла постоянно вносить изменения в стандартные файлы правил защиты, поэтому рекомендуется разместить сначала все полномочия в отдельном файле, например `MyApp.policy`, а затем указать явным образом имя этого файла для каждой прикладной программы. А применить правила защиты можно двумя способами. Во-первых, установить свойства системы в главном методе прикладной программы:

```
System.setProperty("java.security.policy", "MyApp.policy");
```

И во-вторых, запустить виртуальную машину, введя следующую команду:

```
java -Djava.security.policy=MyApp.policy MyApp
```

Для аплетов вместо этого следует ввести такую команду:

```
appletviewer -J-Djava.security.policy=MyApplet.policy MyApplet.html
```

(С помощью параметра `-J` утилита `appletviewer` виртуальной машине Java можно передать любой аргумент из командной строки.)

В приведенных выше примерах файл `MyApp.policy` добавляется к остальным действующим правилам защиты. Но если добавить еще один знак равенства, т.е. ввести команду

```
java -Djava.security.policy==MyApp.policy MyApp
```

то в прикладной программе будет использоваться только указанный файл правил защиты, а все стандартные файлы правил защиты будут игнорироваться.



Внимание! Во время тестирования можно легко допустить ошибку, случайно оставив в текущем каталоге файл `.java.policy`, который может предоставлять многие, а то и все возможные полномочия `[AllPermission]`. Поэтому, заметив, что приложение как будто не обращает внимания на ограничения, указанные в добавленном файле правил защиты, нужно прежде всего проверить, не остался ли в текущем каталоге файл `.java.policy`. Вероятность допустить подобную ошибку наиболее велика в системах UNIX, где файлы, имена которых начинаются с точки, по умолчанию не отображаются.

Как упоминалось ранее, диспетчер защиты в прикладных программах на Java по умолчанию не устанавливается. Это означает, что до тех пор, пока он не будет установлен, увидеть действие файлов правил защиты не удастся. А установить диспетчер защиты можно двумя способами. Во-первых, добавить в тело метода `main()` следующую строку кода:

```
System.setSecurityManager(new SecurityManager());
```

И во-вторых, указать параметр `-Djava.security.manager` при запуске виртуальной машины Java из командной строки:

```
java -Djava.security.manager -Djava.security.policy=MyApp.policy MyApp
```

Далее в этом разделе подробно рассматриваются способы описания полномочий в файле правил защиты и его формата, кроме способов указания сертификатов, речь о которых пойдет ниже. Итак, в файле правил защиты содержится последовательность записей `grant`, каждая из которых имеет следующий вид:

```
grant источник_кода
{
    полномочие1;
    полномочие2;
    ...
}
```

В источнике кода содержатся сведения о кодовой базе (которая может опускаться, если данная запись `grant` распространяется на код из всех источников), а также имена доверенных уполномоченных и подписавших сертификаты лиц (которые могут опускаться, если подписи не являются обязательными для данной записи). Сведения о кодовой базе указываются следующим образом:

```
codeBase "url"
```

Наличие знака косой черты (`/`) в конце URL означает, что ссылка делается на каталог, а отсутствие этого знака — что ссылка делается на архивный JAR-файл, как показано в приведенных ниже примерах.

```
grant codeBase "www.horstmann.com/classes/" { . . . };
grant codeBase "www.horstmann.com/classes/MyApp.jar" { . . . };
```

В качестве разделителя файлов в URL кодовой базы должны всегда использоваться знаки косой черты, даже если речь идет об URL со ссылками на файлы в операционной системе Windows, как в следующем примере:

```
grant codeBase "file:C:/myapps/classes/" { . . . };
```



На заметку! Как известно, URL типа HTTP всегда начинаются двумя знаками косой черты (`http://`). Но в отношении URL типа `file`, по-видимому, возникает путаница из-за того, что средство чтения файлов правил защиты разрешает использовать для них два формата: `file://локальный_файл` и `file:локальный_файл`. Более того, последняя косая черта перед именем диска в Windows не является обязательной. Это означает, что приемлемыми считаются все следующие варианты:

```
file:C:/dir/filename.ext
file:/C:/dir/filename.ext
file://C:/dir/filename.ext
file:///C:/dir/filename.ext
```

Как показывают проведенные нами проверки, на самом деле вариант `file:///C:/dir/filename.ext` также приемлем, но найти этому разумное объяснение нам так и не удалось.

Полномочия задаются в следующей синтаксической форме:

`permission ИмяКласса ИмяЦелевогоОбъекта, СписокДействий;`

Вместо *ИмяКласса* указывается полностью уточненное имя класса, представляющего конкретные полномочия (например, `java.io.FilePermission`), а вместо *ИмяЦелевогоОбъекта* — конкретный целевой объект, на который должно распространяться действие указываемых полномочий. Так, для полномочий доступа к файлам это может быть имя файла и каталога, а для полномочий доступа к сокетам — название и номер порта хоста. И наконец, вместо параметра *СписокДействий* указывается перечень охватываемых данным полномочием допустимых действий (например, чтения или установления соединения), разделяемых запятой. В некоторых классах полномочий не требуется указывать ни имена целевых объектов, ни списки действий. В табл. 9.2 перечислены некоторые наиболее употребительные классы полномочий и охватываемые ими действия.

Таблица 9.2. Классы полномочий и связанные с ними целевые объекты и действия

Полномочия	Целевые объекты	Действия
<code>java.io.FilePermission</code>	Файлы (см. описание в тексте)	<code>read, write, execute, delete</code>
<code>java.net.SocketPermission</code>	Сокеты (см. описание в тексте)	<code>accept, connect, listen, resolve</code>
<code>java.util.PropertyPermission</code>	Свойства (см. описание в тексте)	<code>read, write</code>
<code>java.lang.RuntimePermission</code>	<code>createClassLoader,</code> <code>getClassLoader,</code> <code>setContextClassLoader, enableContextClassLoaderOverride,</code> <code>createSecurityManager,</code> <code>setSecurityManager, exitVM,</code> <code>getenv.variableName,</code> <code>shutdownHooks, setFactory,</code> <code>setIO, modifyThread,</code> <code>stopThread, modifyThreadGroup,</code> <code>getProtectionDomain,</code> <code>readFileDescriptor,</code> <code>writeFileDescriptor,</code> <code>loadLibrary.libraryName,</code> <code>accessClassInPackage.</code> <code>packageName,</code> <code>defineClassInPackage.</code> <code>packageName,</code> <code>accessDeclaredMembers.</code> <code>className, queuePrintJob,</code> <code>getStackTrace, setDefaultUncaughtExceptionHandler,</code> <code>preferences, usePolicy</code>	Отсутствуют

Продолжение табл. 9.2

Полномочия	Целевые объекты	Действия
java.awt.AWTPermission	showWindowWithoutWarningBanner, accessClipboard, accessEventQueue, createRobot, fullScreenExclusive, listenToAllAWEEvents, readDisplayPixels, replaceKeyboardFocusManager, watchMousePointer, setWindowAlwaysOnTop, setAppletStub	Отсутствуют
java.net.NetPermission	setDefaultAuthenticator, specifyStreamHandler, requestPasswordAuthentication, setProxySelector, getProxySelector, setCookieHandler, getCookieHandler, setResponseCache, getResponseCache	Отсутствуют
java.lang.reflect.ReflectPermission	suppressAccessChecks	Отсутствуют
java.io.SerializablePermission	enableSubclassImplementation, enableSubstitution	Отсутствуют
java.security.SecurityPermission	createAccessControlContext, getDomainCombiner, getPolicy, setPolicy, getProperty, keyName, setProperty, keyName, insertProvider, providerName, removeProvider, providerName, setSystemScope, setIdentityPublicKey, setIdentityInfo, addIdentityCertificate, removeIdentityCertificate, printIdentity, clearProviderProperties, providerName, putProviderProperty, providerName, removeProviderProperty, providerName, getSignerPrivateKey, setSignerKeyPair	Отсутствуют
java.security.AllPermission	Отсутствуют	Отсутствуют
javax.audio.AudioPermission	Воспроизведение записи	Отсутствуют

Окончание табл. 9.2

Полномочия	Целевые объекты	Действия
<code>javax.security.auth.AuthPermission</code>	<code>doAs, doAsPrivileged, getSubject, getSubjectFromDomainCombiner, setReadOnly, modifyPrincipals, modifyPublicCredentials, modifyPrivateCredentials, refreshCredential, destroyCredential, createLoginContext.contextName, getLoginConfiguration, setLoginConfiguration, refreshLoginConfiguration</code>	Отсутствуют
<code>java.util.logging.LoggingPermission</code>	<code>control</code>	Отсутствуют
<code>java.sql.SQLPermission</code>	<code>setLog</code>	Отсутствуют

Как следует из табл. 9.2, большинство полномочий просто позволяют выполнять определенные операции. К операции можно относиться как к целевому объекту с подразумеваемым действием "permit" (разрешить). Все классы полномочий, перечисленные в табл. 9.2, являются производными от класса `BasicPermission` (см. рис. 9.7). Но классы с такими целевыми объектами, как файл, сокет и свойство, оказываются более сложными и поэтому заслуживают дополнительного рассмотрения.

Целевые объекты прав доступа к файлам могут иметь следующий вид.

файл	Файл
каталог/	Каталог
каталог/*	Все файлы из данного каталога
*	Все файлы из текущего каталога
каталог/-	Все файлы из данного каталога и всех его подкаталогов
-	Все файлы из текущего каталога и всех его подкаталогов
<<ALL FILES>>	Все файлы из файловой системы

Например, приведенная ниже запись полномочий означает, что доступ разрешается ко всем файлам в каталоге `/myapp` и любом из его подкаталогов.

```
permission java.io.FilePermission "/myapp/-", "read,write,delete";
```

Для обозначения обратной косой черты в пути к файлам в Windows этот знак нужно повторить дважды:

```
permission java.io.FilePermission "c:\\\\myapp\\\\-", "read,write,delete";
```

Целевые объекты прав доступа через сокет требуют указания хоста (т.е. сетевого узла) и диапазона портов. Синтаксическая форма для указания хоста выглядит следующим образом.

имя_хоста или IP-адрес	Одиночный узел
localhost или пустая строка	Локальный узел
*.суффикс_домена	Любой узел, принадлежащий домену, имя которого оканчивается указанным суффиксом
*	Все узлы

Номера портов являются необязательными и указываются в приведенной ниже форме.

- :п** Единственный порт
- :п-** Все порты с номерами **п** и выше
- п** Все порты с номерами **п** и ниже
- :п1-п2** Все порты в пределах от **п1** до **п2**

Ниже приведен пример записи прав доступа через сокет.

```
permission java.net.SocketPermission
    "* .horstmann.com:8000-8999", "connect";
```

И наконец, целевые объекты прав доступа к свойствам могут принимать одну из двух следующих форм.

свойство	Отдельное свойство
предфикс_свойства.*	Все свойства с указанным суффиксом

Таким образом, права доступа к свойствам могут выглядеть как "java.home" и как "java.vm.*". Например, следующая запись полномочий означает, что программе разрешается считывать все свойства, начинающиеся с java.vm:

```
permission java.util.PropertyPermission "java.vm.*", "read";
```

В файлах правил защиты допускается использовать и системные свойства. Лексема \${свойство} в этом случае заменяется значением свойства. Например, лексема \${user.home} заменяется основным каталогом пользователя. Ниже приведен типичный пример применения этого системного свойства в записи полномочий.

```
permission java.io.FilePermission "${user.home}", "read,write";
```

Чтобы упростить дело при составлении не зависящих от используемой платформы файлов правил защиты, вместо явных разделителей / или \\ рекомендуется использовать свойство file.separator или даже его сокращенный вариант \${/}. Например, для предоставления прав на чтение и запись в основном каталоге пользователя и всех его подкаталогах можно воспользоваться следующей записью, удобной с точки зрения переносимости:

```
permission java.io.FilePermission "${user.home}${/-}", "read,write";
```



На заметку! В состав JDK входит примитивная утилита **policytool**, предназначенная для редактирования файлов правил защиты (рис. 9.8). Очевидно, что для конечных пользователей эта утилита не подходит, поскольку большинство ее параметров будет для них совершенно непонятным. На наш взгляд, она больше подходит для системных администраторов, которые предпочитают указывать нужный параметр и щелкать кнопкой мыши, а не вводить сложную синтаксическую конструкцию вручную. Тем не менее этой утилите недостает понятного набора уровней защиты (например, низкого, среднего и высокого), что может быть очень важно для неспециалистов. В общем, мы считаем, что платформа Java содержит все компоненты, требующиеся для создания подробной модели защиты, но в то же время на этой платформе могли бы быть предусмотрены и более удобные средства для работы с подобными моделями как для конечных пользователей, так и для системных администраторов.

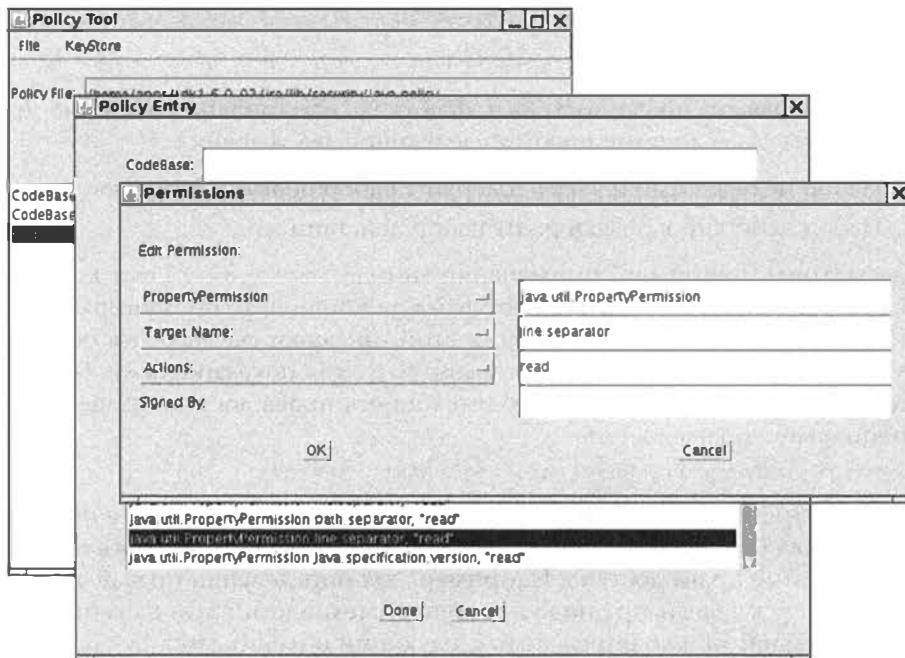


Рис. 9.8. Утилита policytool

9.2.4. Пользовательские полномочия

В этом разделе будет показано, каким образом создаются собственные классы полномочий, на которые пользователи могут ссылаться в файлах правил защиты. Для реализации классов полномочий следует расширить класс `Permission` и реализовать перечисленные ниже методы.

- Конструктор с двумя строковыми параметрами для указания объекта и списка действий.
- Метод `String getActions()`.
- Метод `boolean equals(Object other)`.
- Метод `int hashCode()`.
- Метод `boolean implies(Permission other)`.

Последний метод самый важный. Для полномочий предусмотрен определенный порядок, в соответствии с которым наиболее общие полномочия предполагают использование специальных прав доступа. Например, приведенное ниже право доступа к файлу разрешает чтение и запись любого файла из каталога `/tmp` и любых его подкаталогов.

```
p1 = new FilePermission("/tmp/-", "read, write");
```

Это общее право доступа предполагает наличие других, специальных прав доступа:

```
p2 = new FilePermission("/tmp/-", "read");
p3 = new FilePermission("/tmp/aFile", "read, write");
p4 = new FilePermission("/tmp/aDirectory/-", "write");
```

Иными словами, право доступа к файлу p1 предполагает наличие другого права доступа p2, если выполняются следующие два условия.

1. Набор целевых файлов в p1 содержит набор целевых файлов в p2.
2. Набор действий в p1 содержит набор действий в p2.

Рассмотрим, например, применение метода `implies()`. Если конструктор класса `FileInputStream()` открывает файл для чтения, то он проверяет наличие прав такого доступа. Для выполнения этой проверки специальный объект прав доступа передается методу `checkPermission()`, как показано ниже. После этого диспетчер защиты запрашивает все имеющиеся права доступа, предполагается ли специальное право доступа.

```
checkPermission(new FilePermission(fileName, "read"));
```

В частности, полномочия типа `AllPermission` предполагают все права доступа. Определяя собственные классы полномочий, необходимо также определить предполагаемые права доступа. Например, для определения прав доступа пользователя Томму к телевизору под управлением технологии Java в некотором промежутке времени можно определить следующий объект класса `TVPermission`:

```
new TVPermission("Tommy:2-12:1900-2200", "watch, record")
```

Этот класс разрешает пользователю Томму смотреть и записывать телевизионные передачи на каналах 2–12 в период времени от 19:00 до 22:00. А для применения приведенного ниже специального права доступа придется реализовать метод `implies()`.

```
new TVPermission("Tommy:4:2000-2100", "watch")
```

9.2.5. Реализация класса полномочий

В этом разделе на примере конкретной программы демонстрируется реализация новых полномочий для контроля над текстом, вставляемым в текстовую область. В обязанности этой программы входит предотвращение попыток ввода в текстовую область всевозможных “непристойных слов”, вроде `sex`, `drugs` и `C++`, но не `rock-n-roll!` Чтобы предоставить возможность передавать список всех подобных непристойных слов в файл правил защиты, применяется специальный класс полномочий. Он является производным от класса `JTextArea` и всегда запрашивает диспетчера защиты, можно ли вводить новый текст в текстовую область, как показано ниже. Если диспетчер защиты предоставляет полномочия типа `WordCheckPermission`, новый текст вводится, а иначе метод `checkPermission()` генерирует исключение.

```
class WordCheckTextArea extends JTextArea
{
    public void append(String text)
    {
        WordCheckPermission p = new WordCheckPermission(text, "insert");
        SecurityManager manager = System.getSecurityManager();
```

```

if (manager != null) manager.checkPermission(p);
super.append(text);
}
}

```

Полномочия на проверку слов (`WordCheckPermission`) допускают выполнение двух следующих действий: `insert` (вставка указанного текста) и `avoid` (ввод текста без указанных непристойных слов). Запускать рассматриваемую здесь программу следует с помощью приведенного ниже файла правил защиты. В этом файле предоставляется разрешение на вставку любого текста, который не содержит такие непристойные слова, как `sex`, `drugs` и `C++`.

```

grant
{
    permission permissions.WordCheckPermission
        "sex,drugs,C++", "avoid";
};

```

При разработке класса `WordCheckPermission` особое внимание следует уделить методу `implies()`. Ниже перечислены правила, определяющие, должны ли полномочия `p1` предполагать полномочия `p2`.

- Если полномочия `p1` допускают действие `avoid`, а полномочия `p2` — действие `insert`, то целевой объект с полномочиями `p2` должен исключать все слова из полномочий `p1`. Например, следующие полномочия:
`permissions.WordCheckPermission "sex, drugs, C++", "avoid"`
- предполагают такие полномочия:
`permissions.WordCheckPermission "Mary had a little lamb", "insert"`
- Если оба вида полномочий `p1` и `p2` допускают действие `avoid`, то набор слов с полномочиями `p2` должен содержать все слова из набора слов с полномочиями `p1`. Например, следующие полномочия:
`permissions.WordCheckPermission "sex,drugs", "avoid"`
- предполагают такие полномочия:
`permissions.WordCheckPermission "sex,drugs,C++", "avoid".`
- Если оба вида полномочий `p1` и `p2` допускают действие `insert`, то текст с полномочиями `p1` должен содержать текст с полномочиями `p2`. Например, следующие полномочия:
`permissions.WordCheckPermission "Mary had a little lamb", "insert"`
- предполагают такие полномочия:
`permissions.WordCheckPermission "a little lamb", "insert"`

Исходный код, реализующий класс `WordCheckPermission`, представлен в листинге 9.4. Следует заметить, что целевой объект полномочий извлекается методом с не совсем подходящим именем `getName()` из класса `Permission`.

Полномочия описываются в файлах правил защиты с помощью пары символьных строк, поэтому классы прав доступа должны быть подготовлены к синтаксическому анализу этих строк. Так, в рассматриваемом здесь примере для преобразования списка разделяемых запятыми непристойных слов с полномочиями `avoid` в подлинное множество типа `Set` используется следующий метод:

```
public Set<String> badWordSet()
{
    Set<String> set = new HashSet<String>();
    set.addAll(Arrays.asList(getName().split(", ")));
    return set;
}
```

Этот метод позволяет использовать для сравнения множеств (в данном случае наборов слов) методы `equals()` и `containsAll()`. Как было показано в главе 3, метод `equals()` из класса `Set` признает два множества равными в том случае, если в них содержатся одинаковые элементы, независимо от порядка их расположения. Например, наборы слов "sex, drugs, C++" и "C++, drugs, sex" будут признаны равными.

Внимание! Класс полномочий должен быть непременно открытим (`public`). Загрузчик файлов правил защиты не может загружать классы с уровнем доступности пакета, выходящим за пределы доступности пути к классам начальной загрузки, и поэтому он просто игнорирует любые классы, которые ему не удается обнаружить.

В исходном коде, приведенном в листинге 9.5, показано, каким образом действует класс `WordCheckPermission`. Попробуйте ввести в текстовой области какой-нибудь текст и щелкнуть на кнопке `Insert` (Вставить). Если проверка на безопасность пройдет успешно, текст будет вставлен в текстовую область, в противном случае появится сообщение об ошибке (рис. 9.9).

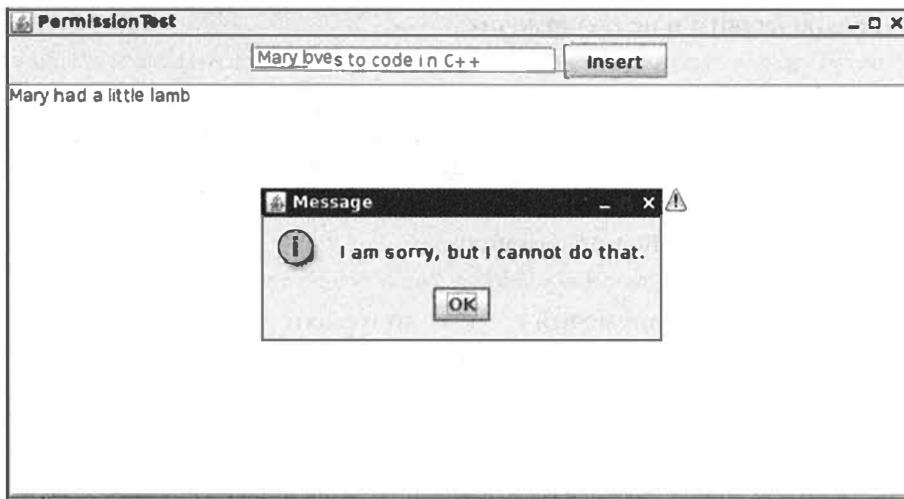


Рис. 9.9. Программа PermissionTest в действии

Внимание! Глядя на рис. 9.9, можно заметить маленький треугольник в окне сообщения, предупреждающий о том, что данное окно могло появиться без разрешения. Некогда грозное сообщение "**Untrusted Java Applet Window**" (Окно не заслуживающего доверия аплета) в последующих версиях JDK было смягчено и стало, по существу, бесполезным для предупреждения пользователей. Это предупреждение можно отключить с помощью целевого объекта `showWindowWithoutWarningBanner` полномочий типа `java.awt.AWTPermission`. Чтобы предоставить эти полномочия, достаточно отредактировать файл правил защиты.

На этом рассмотрение способов и средств настройки безопасности на платформе Java завершается. Как правило, вам придется лишь подкорректировать должным образом стандартные полномочия. Но если потребуются дополнительные средства для контроля безопасности, то вы сможете теперь определить специальные классы полномочий и настроить их таким же образом, как и стандартные полномочия.

Листинг 9.4. Исходный код из файла permissions/WordCheckPermission.java

```
1 package permissions;
2
3 import java.security.*;
4 import java.util.*;
5
6 /**
7  * Полномочия на проверку непристойных слов
8 */
9 public class WordCheckPermission extends Permission
10 {
11     private String action;
12
13     /**
14      * Конструирует полномочия на проверку непристойных слов
15      * @param target Список разделяемых запятыми слов
16      * @param anAction Действие "вставить" или "исключить"
17     */
18     public WordCheckPermission(String target, String anAction)
19     {
20         super(target);
21         action = anAction;
22     }
23     public String getActions()
24     {
25         return action;
26     }
27
28     public boolean equals(Object other)
29     {
30         if (other == null) return false;
31         if (!getClass().equals(other.getClass())) return false;
32         WordCheckPermission b = (WordCheckPermission) other;
33         if (!Objects.equals(action, b.action)) return false;
34         if ("insert".equals(action)) return
35             Objects.equals(getName(), b.getName());
36         else if ("avoid".equals(action)) return
37             badWordSet().equals(b.badWordSet());
38         else return false;
39     }
40
41     public int hashCode()
42     {
43         return Objects.hash(getName(), action);
44     }
45
46     public boolean implies(Permission other)
47     {
```

```

48     if (!(other instanceof WordCheckPermission)) return false;
49     WordCheckPermission b = (WordCheckPermission) other;
50     if (action.equals("insert"))
51     {
52         return b.action.equals("insert") &&
53             getName().indexOf(b.getName()) >= 0;
54     }
55     else if (action.equals("avoid"))
56     {
57         if (b.action.equals("avoid")) return
58             b.badWordSet().containsAll(badWordSet());
59         else if (b.action.equals("insert"))
60         {
61             for (String badWord : badWordSet())
62                 if (b.getName().indexOf(badWord) >= 0) return false;
63             return true;
64         }
65         else return false;
66     }
67     else return false;
68 }
69 /**
70  * Получает непристойные слова, описываемые данным
71  * правилом прав доступа
72  * @return Возвращает непристойные слова
73 */
74 public Set<String> badWordSet()
75 {
76     Set<String> set = new HashSet<>();
77     set.addAll(Arrays.asList(getName().split(",")));
78     return set;
79 }
80 }
81 }
```

Листинг 9.5. Исходный код из файла permissions/PermissionTest.java

```

1 package permissions;
2
3 import java.awt.*;
4
5 import javax.swing.*;
6
7 /**
8  * Этот класс демонстрирует применение специальных полномочий
9  * типа WordCheckPermission
10 * @version 1.04 2016-05-10
11 * @author Cay Horstmann
12 */
13 public class PermissionTest
14 {
15     public static void main(String[] args)
16     {
17         System.setProperty("java.security.policy",
18                           "permissions/PermissionTest.policy");
19         System.setSecurityManager(new SecurityManager());
20         EventQueue.invokeLater(() ->
```

```
21     {
22         JFrame frame = new PermissionTestFrame();
23         frame.setTitle("PermissionTest");
24         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25         frame.setVisible(true);
26     });
27 }
28 /**
29  * Этот фрейм содержит текстовое поле для ввода слов в текстовую
30  * область, защищенную от вставки непристойных слов
31  */
32 class PermissionTestFrame extends JFrame
33 {
34     private JTextField textField;
35     private WordCheckTextArea textArea;
36     private static final int TEXT_ROWS = 20;
37     private static final int TEXT_COLUMNS = 60;
38
39     public PermissionTestFrame()
40     {
41         textField = new JTextField(20);
42         JPanel panel = new JPanel();
43         panel.add(textField);
44         JButton openButton = new JButton("Insert");
45         panel.add(openButton);
46         openButton.addActionListener(event ->
47             insertWords(textField.getText()));
48
49         add(panel, BorderLayout.NORTH);
50
51         textArea = new WordCheckTextArea();
52         textArea.setRows(TEXT_ROWS);
53         textArea.setColumns(TEXT_COLUMNS);
54         add(new JScrollPane(textArea), BorderLayout.CENTER);
55         pack();
56     }
57 /**
58  * Пытается вставить слова в текстовую область. Отображает
59  * диалоговое окно, если попытка окажется неудачной
60  * @param words Вставляемые слова
61  */
62 public void insertWords(String words)
63 {
64     try
65     {
66         textArea.append(words + "\n");
67     }
68     catch (SecurityException ex)
69     {
70         JOptionPane.showMessageDialog(this,
71             "I am sorry, but I cannot do that.");
72         ex.printStackTrace();
73     }
74 }
75 }
```

```
79
80 /**
81  * Текстовая область, в которой метод ввода текста
82  * проверяет в целях безопасности, чтобы в нее не были
83  * вставлены непристойные слова
84 */
85 class WordCheckTextArea extends JTextArea
86 {
87     public void append(String text)
88     {
89         WordCheckPermission p = new WordCheckPermission(
90             text, "insert");
91         SecurityManager manager = System.getSecurityManager();
92         if (manager != null) manager.checkPermission(p);
93         super.append(text);
94     }
95 }
```

java.security.Permission 1.2

- **Permission(String name)**
Создает полномочия с указанным именем целевого объекта.
- **String getName()**
Возвращает имя целевого объекта для данных полномочий.
- **boolean implies(Permission other)**
Проверяет, предполагают ли данные полномочия другие полномочия. Это имеет место в том случае, если в других полномочиях описывается более конкретное условие, вытекающее из условия, указанного в данных полномочиях.

9.3. Аутентификация пользователей

В прикладном программном интерфейсе Java API предоставляется каркас JAAS (Java Authentication and Authorization Service — Служба аутентификации и авторизации в Java). Он позволяет сочетать аутентификацию, предоставляемую на отдельной платформе, с управлением правами доступа и подробно рассматривается в последующих разделах.

9.3.1. Каркас JAAS

Как следует из названия каркаса JAAS, он состоит из двух компонентов. В частности, компонент аутентификации отвечает за опознавание пользователей программ, а компонент авторизации — за проверку их полномочий.

Каркас JAAS, по существу, представляет собой встраиваемый прикладной программный интерфейс API, отделяющий прикладные программы на Java от конкретной технологии, применяемой для реализации средств аутентификации. Помимо прочего, в нем поддерживаются механизмы регистрации в UNIX и NT, а также механизмы аутентификации Kerberos и по сертификатам.

После аутентификации за пользователем может быть закреплен определенный набор полномочий. Ниже приведен пример, в котором пользователю *harry*

предоставляется особый ряд полномочий, которых нет ни у кого из других пользователей.

```
grant principal com.sun.security.auth.UnixPrincipal "harry"
{
    permission java.util.PropertyPermission "user.*", "read";
    ...
};
```

В данном примере класс `com.sun.security.auth.UnixPrincipal` выполняет проверку имени пользователя UNIX, запускающего программу. Метод `getName()` из этого класса возвращает имя пользователя, которое зарегистрировано в системе UNIX и сравнивается на равенство с именем `harry`.

Класс `LoginContext` дает диспетчеру защиты возможность проверить правильность предоставления таких полномочий. Ниже приведена общая структура кода регистрации.

```
try
{
    System.setSecurityManager(new SecurityManager());
    LoginContext context = new LoginContext("Login1");
    // определяется в конфигурационном файле JAAS
    context.login();
    // получить аутентифицированный объект типа Subject
    Subject subject = context.getSubject();
    ...
    context.logout();
}
catch (LoginException exception)
{
    // это исключение генерируется при неудачной попытке регистрации
    exception.printStackTrace();
}
```

Теперь объект `subject` представляет прошедшего аутентификацию пользователя. Строковый параметр `"Login1"` в конструкторе класса `LoginContext` обозначает запись с аналогичным именем в конфигурационном файле JAAS. Ниже приведен пример такого конфигурационного файла.

```
Login1
{
    com.sun.security.auth.module.UnixLoginModule required;
    com.whizzbang.auth.module.RetinaScanModule sufficient;
};

Login2
{
    ...
};
```

Разумеется, в JDK не предусмотрено никаких модулей для биометрической регистрации. В состав пакета `com.sun.security.auth.module` входят только следующие модули:

```
UnixLoginModule
NTLoginModule
Krb5LoginModule
JndiLoginModule
KeyStoreLoginModule
```

Правила регистрации состоят из ряда модулей, каждый из которых обозначен как required, sufficient, requisite или optional. Значение этих ключевых слов можно понять из приведенного ниже описания алгоритма регистрации.

При регистрации выполняется аутентификация *субъекта*, который может иметь несколько *принципалов*. Каждый принципал просто описывает какое-то свойство субъекта, например, имя пользователя, идентификатор группы или роль. Как было показано ранее в операторе grant, принципалы управляют правами доступа. Объект типа com.sun.security.auth.UnixPrincipal описывает имя пользователя, зарегистрированное в UNIX, а объект типа UnixNumeric Group Principal может выполнять проверку на принадлежность пользователя к группе в UNIX. Предложение grant дает возможность проверить наличие принципала с помощью следующей синтаксической конструкции:

```
grant КлассПринципала "ИмяПринципала"
```

Следовательно, в рассматриваемом здесь примере это предложение принимает следующий вид:

```
grant com.sun.security.auth.UnixPrincipal "harry"
```

Когда пользователь регистрируется, в отдельном контексте управления доступом должен быть выполнен код, требующий проверки принципалов. Для инициализации нового привилегированного действия типа PrivililegedAction вызывается статический метод doAs() или doAsPrivileged().

Оба эти метода выполняют действие, вызывая метод run() для объекта, класс которого реализует интерфейс PrivililegedAction, а также используя полномочия принципалов субъекта, как показано ниже. Если же при выполнении действий генерируются проверяемые исключения, то вместо упомянутого выше интерфейса лучше реализовать интерфейс PrivililegedExceptionAction.

```
PrivililegedAction<T> action = () ->
{
    // выполнить код с полномочиями принципалов субъекта
    . . .
};

T result = Subject.doAs(subject, action);
// или Subject.doAsPrivileged(subject, action, null)
```

Методы doAs() и doAsPrivileged() отличаются лишь незначительно. Так, метод doAs() запускается в текущем контексте управления доступом, а метод doAsPrivileged() — в новом контексте. Кроме того, метод doAsPrivileged() позволяет разделять права доступа для кода регистрации и бизнес-логики. В рассматриваемом здесь примере прикладной программы код регистрации имеет следующие полномочия:

```
permission javax.security.auth.AuthPermission
        "createLoginContext.Login1";
permission javax.security.auth.AuthPermission "doAsPrivileged";
```

Пользователь, прошедший аутентификацию, получает приведенные ниже полномочия. Если бы вместо метода doAsPrivileged() использовался метод doAs(), такие же полномочия требовались бы и коду регистрации!

```
permission java.util.PropertyPermission "user.*", "read";
```

В рассматриваемом здесь примере программы AuthTest (ее исходный код приведен в листингах 9.6 и 9.7) демонстрируется, как ограничивать права доступа определенных пользователей. Эта программа аутентифицирует пользователя, а затем выполняет действие, извлекающее системное свойство. Чтобы данный пример программы оказался работоспособным, коды регистрации и действия следует упаковать в два отдельных архивных JAR-файла, выполнив следующие команды:

```
javac auth/*.java  
jar cvf login.jar auth/AuthTest.class  
jar cvf action.jar auth/SysPropAction.class
```

Как следует из файла правил защиты, содержимое которого приведено в листинге 9.8, пользователь системы UNIX с именем `harry` обладает правами на чтение всех файлов. Замените сначала имя `harry` своим учетным именем, а затем выполните приведенную ниже команду. Настройка регистрации представлена в листинге 9.9.

```
java -classpath login.jar:action.jar  
-Djava.security.policy=auth/AuthTest.policy  
-Djava.security.auth.login.config=auth/jaas.config  
auth.AuthTest
```

В системе Windows нужно заменить `UnixPrincipal` на `NTUserPrincipal` в файлах `AuthTest.policy` и `jaas.configUnix`, а также использовать точку с запятой для разделения архивных JAR-файлов, как показано в приведенной ниже команде.

```
java -classpath login.jar;action.jar . . .
```

После этого программа `AuthTest` должна отобразить значение свойства `user.home`. Но если зарегистрироваться под другим учетным именем, то должно возникнуть исключение в связи с отсутствием требующихся полномочий.

Внимание! Очень важно выполнить все приведенные выше инструкции точно и аккуратно, поскольку даже незначительные отклонения могут привести к неверной настройке аутентификации пользователей.

Листинг 9.6. Исходный код из файла auth/AuthTest.java

```
1 package auth;  
2  
3 import java.security.*;  
4 import javax.security.auth.*;  
5 import javax.security.auth.login.*;  
6  
7 /**  
8  * В этой программе демонстрируется аутентификация  
9  * пользователей через специальную регистрацию и  
10 * последующее выполнение действия типа SysPropAction  
11 * привилегиями зарегистрированного пользователя  
12 * @version 1.01 2007-10-06  
13 * @author Cay Horstmann  
14 */  
15 public class AuthTest  
16 {
```

```

17 public static void main(final String[] args)
18 {
19     System.setSecurityManager(new SecurityManager());
20     try
21     {
22         LoginContext context = new LoginContext("Login1");
23         context.login();
24         System.out.println("Authentication successful.");
25         Subject subject = context.getSubject();
26         System.out.println("subject=" + subject);
27         PrivilegedAction<String> action = new SysPropAction("user.home");
28         String result = Subject.doAsPrivileged(subject, action, null);
29         System.out.println(result);
30         context.logout();
31     }
32     catch (LoginException e)
33     {
34         e.printStackTrace();
35     }
36 }
37 }
```

Листинг 9.7. Исходный код из файла auth/SysPropAction.java

```

1 package auth;
2
3 import java.security.*;
4
5 /**
6  * Это действие осуществляет поиск системного свойства
7  * @version 1.01 2007-10-06
8  * @author Cay Horstmann
9 */
10 public class SysPropAction implements PrivilegedAction<String>
11 {
12     private String propertyName;
13
14     /**
15      * Конструирует действие для поиска заданного свойства
16      * @param propertyName Имя свойства (например, "user.home")
17      */
18     public SysPropAction(String propertyName)
19     { this.propertyName = propertyName; }
20
21     public String run()
22     {
23         return System.getProperty(propertyName);
24     }
25 }
```

Листинг 9.8. Исходный код из файла auth/AuthTest.policy

```

1 grant codebase "file:login.jar"
2 {
3     permission javax.security.auth.AuthPermission
4             "createLoginContext.Login1";
5     permission javax.security.auth.AuthPermission
6             "doAsPrivileged";
```

```
7  };
8
9 grant principal com.sun.security.auth.UnixPrincipal "harry"
10 {
11   permission java.util.PropertyPermission "user.*", "read";
12 };
```

Листинг 9.9. Исходный код из файла auth/jaas.config

```
1 Login1
2 {
3   com.sun.security.auth.module.UnixLoginModule required;
4 };
```

javax.security.auth.login.LoginContext 1.4

- **LoginContext(String name)**

Создает контекст регистрации. Параметр **name** соответствует дескриптору регистрации в конфигурационном файле службы JAAS.

- **void login()**

Регистрирует субъект, а при неудачном исходе регистрации генерирует исключение типа **LoginException**. Вызывает метод **login()** для диспетчеров, указанных в конфигурационном файле JAAS.

- **void logout()**

Отменяет регистрацию субъекта. Вызывает метод **logout()** для диспетчеров, указанных в конфигурационном файле JAAS.

- **Subject getSubject()**

Возвращает аутентифицированный субъект.

javax.security.auth.Subject 1.4

- **Set<Principal> getPrincipals()**

Возвращает принципалы данного субъекта.

- **static Object doAs(Subject subject, PrivilegedAction action)**

- **static Object doAs(Subject subject, PrivilegedExceptionAction action)**

- **static Object doAsPrivileged(Subject subject, PrivilegedAction action, AccessControlContext context)**

- **static Object doAsPrivileged(Subject subject, PrivilegedExceptionAction action, AccessControlContext context)**

Выполняют привилегированное действие от имени субъекта. Возвращают объект, сформированный методом **run()** из класса, реализующего интерфейс **PrivilegedAction**. Методы **doAsPrivileged()** выполняют действие в указанном контексте управления доступом. Для них можно указать "моментальный снимок" контекста, полученный ранее в результате вызова метода **AccessController.getContext()**. Если задано пустое значение **null**, код будет выполняться в новом контексте.

java.security.PrivilegedAction 1.4

- **Object run()**

Этот метод необходимо определить самостоятельно для выполнения кода от имени субъекта.

java.security.PrivilegedExceptionAction 1.4

- **Object run()**

Этот метод необходимо определить самостоятельно для выполнения кода от имени субъекта. Он может генерировать любые проверяемые исключения.

java.security.Principal 1.1

- **String getName()**

Возвращает имя данного принципала.

9.3.2. Модули регистрации JAAS

В этом разделе рассматривается пример применения JAAS, демонстрирующий следующее:

- как реализовывать свой собственный модуль регистрации;
- как реализовывать ролевую аутентификацию.

Предоставление собственного модуля регистрации бывает полезным в тех случаях, когда учетные данные сохраняются в базе данных. Даже если вас вполне устраивает стандартный модуль регистрации, изучение процесса реализации специального модуля поможет вам лучше разобраться с предназначением параметров в конфигурационном файле JAAS.

Реализация механизма ролевой аутентификации играет важную роль в тех случаях, когда требуется управлять большим количеством пользователей. Размещать имена всех допустимых пользователей в файле правил регистрации непрактично. Вместо этого лучше поступить таким образом, чтобы модуль регистрации сопоставлял пользователей с ролями вроде "admin" или "HR" и предоставлял им права, исходя из их ролей.

Одна из задач модуля регистрации состоит в заполнении множества принципалов аутентифицируемого субъекта. Если в модуле регистрации поддерживаются роли, он должен также вводить в это множество объекты типа Principal, описывающие соответствующие роли. К сожалению, в библиотеке Java отсутствует класс для решения этой задачи, поэтому нам пришлось создать собственный класс, исходный код которого приведен в листинге 9.10. Этот класс просто сохраняет пары "описание–значение" типа role=admin. А его метод getName() возвращает их, что дает возможность вводить ролевые полномочия в файл правил регистрации, как показано ниже.

```
grant principal SimplePrincipal "role=admin" { . . . }
```

Рассматриваемый здесь модуль регистрации предусматривает поиск имен, паролей и ролей пользователей в текстовом файле, содержащем строки, аналогичные приведенным ниже. Разумеется, при разработке реального модуля регистрации эти сведения следовало бы хранить в какой-нибудь базе данных или словаре.

```
harry|secret|admin  
carl|guessme|HR
```

Исходный код рассматриваемого здесь примера модуля регистрации типа SimpleLoginModule представлен в листинге 9.11. Метод checkLogin() проверяет, соответствуют ли имя пользователя и пароль какой-нибудь записи в файле паролей. Если они соответствуют, то во множество принципалов данного субъекта вводятся два объекта типа SimplePrincipal, как показано ниже.

```
Set<Principal> principals = subject.getPrincipals();  
principals.add(new SimplePrincipal("username", username));  
principals.add(new SimplePrincipal("role", role));
```

Остальная часть модуля типа SimpleLoginModule довольно проста. В частности, метод initialize() получает в качестве параметров следующее.

- Аутентифицируемый объект типа Subject.
- Обработчик для извлечения учетных данных.
- Отображение sharedState, которое можно использовать для обмена данными между модулями регистрации.
- Отображение options, которое содержит пары "имя–значение", задаваемые при настройке модуля регистрации.

Допустим, что рассматриваемый здесь модуль регистрации настраивается приведенным ниже образом. В таком случае этот модуль будет извлекать из отображения options параметры настройки pwfile.

```
SimpleLoginModule required pwfile="password.txt";
```

Сбором имени пользователя и пароля данный модуль регистрации не занимается, поскольку эта задача порученациальному обработчику. Такое разделение ответственности позволяет использовать один и тот же модуль, не особенно беспокоясь, откуда именно поступают учетные данные: из диалогового окна ГПИ, командной строки консоли или конфигурационного файла. Требующийся обработчик задается при создании экземпляра класса LoginContext, как показано в приведенном ниже примере кода.

```
LoginContext context = new LoginContext("Login1",  
    new com.sun.security.auth.callback.DialogCallbackHandler());
```

Обработчик типа DialogCallbackHandler открывает простое диалоговое окно ГПИ для запрашивания имени пользователя и пароля, а обработчик типа com.sun.security.auth.callback.TextCallbackHandler получает эти учетные данные с консоли.

Но в данном примере для получения имени пользователя и пароля используется собственный ГПИ (рис. 9.10). Для этой цели создается специальный обработчик, способный сохранять и возвращать учетные данные, как показано в листинге 9.12.

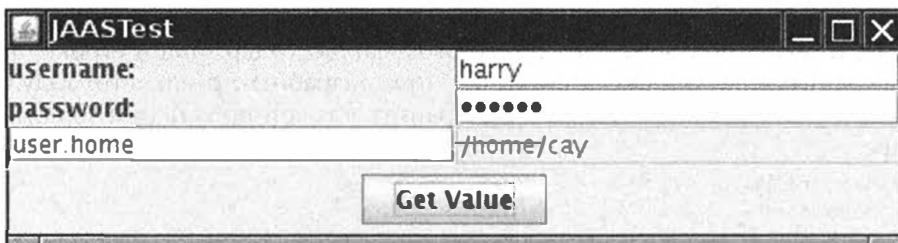


Рис. 9.10. Специальный модуль регистрации

Этот обработчик состоит из единственного метода `handle()`, который обрабатывает массив объектов типа `Callback`. Интерфейс `Callback` реализуется рядом предопределенных классов вроде `NameCallback` и `PasswordCallback`. При желании можно добавить к ним собственный класс, например `RetinaScanCallback`. Исходный код этого обработчика выглядит не очень изящно, поскольку он нуждается в анализе типа объектов обратного вызова, как показано ниже.

```
public void handle(Callback[] callbacks)
{
    for (Callback callback : callbacks)
    {
        if (callback instanceof NameCallback) . . .
        else if (callback instanceof PasswordCallback) . . .
        else . . .
    }
}
```

Модуль регистрации сначала подготавливает массив объектов типа `Callback`, которые требуются ему для аутентификации, как следует из приведенного ниже фрагмента кода, а затем просто извлекает из этих объектов всю необходимую информацию.

```
NameCallback nameCall = new NameCallback("username: ");
PasswordCallback passCall = new PasswordCallback("password: ", false);
callbackHandler.handle(new Callback[] { nameCall, passCall });
```

При выполнении примера программы из листинга 9.1 отображается форма для ввода учетных данных и имени системного свойства. При удачном исходе аутентификации пользователя значение этого свойства извлекается в объект типа `PrivilegedAction`. Как следует из файла правил регистрации, содержимое которого приведено в листинге 9.1, правами на чтение свойств обладают только пользователи с ролью `admin`.

Как и в примере программы из предыдущего раздела, код регистрации и код действия требуется разделить. Для этого создаются два архивных JAR-файла:

```
javac *.java
jar cvf login.jar JAAS*.class Simple*.class
jar cvf action.jar SysPropAction.class
```

После этого рассматриваемая здесь программа запускается на выполнение по приведенной ниже команде. А содержимое конфигурационного файла представлено в листинге 9.1-5.

```
java -classpath login.jar:action.jar
-Djava.security.policy=JAASTest.policy
-Djava.security.auth.login.config=jaas.config
JAASTest
```



На заметку! Можно также обеспечить поддержку более сложного двухэтапного протокола и с его помощью сделать так, чтобы регистрация считалась завершенной только при успешном выполнении всех модулей, предусмотренных в настройке регистрации. Подробнее об этом можно узнать из руководства для разработчиков модулей регистрации, доступного по адресу <http://docs.oracle.com/javase/8/docs/technotes/guides/security/jaas/JAASIMDevGuide.html>.

Листинг 9.10. Исходный код из файла jaas/SimplePrincipal.java

```
1 package jaas;
2
3 import java.security.*;
4 import java.util.*;
5
6 /**
7  * Принципал с именованным значением (например,
8  * "role=HR" или "username=harry")
9 */
10 public class SimplePrincipal implements Principal
11 {
12     private String descr;
13     private String value;
14
15     /**
16      * Конструирует принципал типа SimplePrincipal для
17      * хранения отдельного описания и значения
18      * @param descr Описание
19      * @param value Связанное с ним значение
20     */
21     public SimplePrincipal(String descr, String value)
22     {
23         this.descr = descr;
24         this.value = value;
25     }
26
27     /**
28      * Возвращает имя роли данного принципала
29      * @return Имя роли
30     */
31     public String getName()
32     {
33         return descr + "=" + value;
34     }
35
36     public boolean equals(Object otherObject)
37     {
38         if (this == otherObject) return true;
39         if (otherObject == null) return false;
40         if (getClass() != otherObject.getClass()) return false;
41         SimplePrincipal other = (SimplePrincipal) otherObject;
42         return descr.equals(other.descr) &amp;
```

```

41     return Objects.equals(getName(), other.getName());
42 }
43
44 public int hashCode()
45 {
46     return Objects.hashCode(getName());
47 }
48 }
```

Листинг 9.11. Исходный код из файла jaas/SimpleLoginModule.java

```

1  package jaas;
2
3  import java.io.*;
4  import java.nio.file.*;
5  import java.security.*;
6  import java.util.*;
7  import javax.security.auth.*;
8  import javax.security.auth.callback.*;
9  import javax.security.auth.login.*;
10 import javax.security.auth.spi.*;
11
12 /**
13  * Этот модуль регистрации аутентифицирует пользователей,
14  * считывая их имена, пароли и роли из текстового файла
15 */
16 public class SimpleLoginModule implements LoginModule
17 {
18     private Subject subject;
19     private CallbackHandler callbackHandler;
20     private Map<String, ?> options;
21
22     public void initialize(Subject subject,
23                           CallbackHandler callbackHandler,
24                           Map<String, ?> sharedState,
25                           Map<String, ?> options)
26     {
27         this.subject = subject;
28         this.callbackHandler = callbackHandler;
29         this.options = options;
30     }
31
32     public boolean login() throws LoginException
33     {
34         if (callbackHandler == null)
35             throw new LoginException("no handler");
36
37         NameCallback nameCall = new NameCallback("username: ");
38         PasswordCallback passCall = new PasswordCallback(
39                                     "password: ", false);
40
41         try
42         {
43             callbackHandler.handle(new Callback[]
44                               { nameCall, passCall });
45         }
46         catch (UnsupportedCallbackException e)
47         {
48             LoginException e2 = new LoginException(
49                 "Unsupported callback");
```

```
49         e2.initCause(e);
50         throw e2;
51     }
52     catch (IOException e)
53     {
54         LoginException e2 = new LoginException(
55                             "I/O exception in callback");
56         e2.initCause(e);
57         throw e2;
58     }
59
60     try
61     {
62         return checkLogin(nameCall.getName(),
63                             passCall.getPassword());
64     }
65     catch (IOException ex)
66     {
67         LoginException ex2 = new LoginException();
68         ex2.initCause(ex);
69         throw ex2;
70     }
71 }
72
73 /**
74 * Проверяет достоверность данных аутентификации.
75 * Если они достоверны, то субъекту требуются принципалы
76 * для имени пользователя и его роли
77 * @param username Имя пользователя
78 * @param password Символьный массив, содержащий пароль
79 * @return Возвращает логическое значение true, если
80 *         данные достоверны
81 */
82 private boolean checkLogin(String username, char[] password)
83         throws LoginException, IOException
84 {
85     try (Scanner in = new Scanner(Paths.get(
86             "" + options.get("pwfile")), "UTF-8"))
87     {
88         while (in.hasNextLine())
89         {
90             String[] inputs = in.nextLine().split("\\|");
91             if (inputs[0].equals(username) &&
92                 Arrays.equals(inputs[1].toCharArray(), password))
93             {
94                 String role = inputs[2];
95                 Set<Principal> principals = subject.getPrincipals();
96                 principals.add(new SimplePrincipal(
97                         "username", username));
98                 principals.add(new SimplePrincipal("role", role));
99                 return true;
100            }
101        }
102        return false;
103    }
104 }
105
106 public boolean logout()
107 {
108     return true;
```

```

109 }
110
111 public boolean abort()
112 {
113     return true;
114 }
115
116 public boolean commit()
117 {
118     return true;
119 }
120 }

```

Листинг 9.12. Исходный код из файла aas/SimpleCallbackHandler.java

```

1 package jaas;
2
3 import javax.security.auth.callback.*;
4
5 /**
6  * Этот простой обработчик обратных вызовов предоставляет
7  * заданные имя пользователя и пароль
8  */
9 public class SimpleCallbackHandler implements CallbackHandler
10 {
11     private String username;
12     private char[] password;
13
14     /**
15      * Конструирует обработчик обратных вызовов
16      * @param username Имя пользователя
17      * @param password Символьный массив, содержащий пароль
18     */
19     public SimpleCallbackHandler(String username, char[] password)
20     {
21         this.username = username;
22         this.password = password;
23     }
24
25     public void handle(Callback[] callbacks)
26     {
27         for (Callback callback : callbacks)
28         {
29             if (callback instanceof NameCallback)
30             {
31                 ((NameCallback) callback).setName(username);
32             }
33             else if (callback instanceof PasswordCallback)
34             {
35                 ((PasswordCallback) callback).setPassword(password);
36             }
37         }
38     }
39 }

```

Листинг 9.13. Исходный код из файла jaas/JAASTest.java

```

1 package jaas;
2

```

```

3 import java.awt.*;
4 import javax.swing.*;
5
6 /** В этой программе сначала производится аутентификация
7 * пользователя через специальную регистрацию, а затем
8 * поиск системного свойства с назначеными для
9 * пользователя привилегиями
10 * @version 1.02 2016-05-10
11 * @author Cay Horstmann
12 */
13
14 public class JAASTest
15 {
16     public static void main(final String[] args)
17     {
18         System.setSecurityManager(new SecurityManager());
19         EventQueue.invokeLater(() ->
20         {
21             JFrame frame = new JAASFrame();
22             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23             frame.setTitle("JAASTest");
24             frame.setVisible(true);
25         });
26     }
27 }

```

Листинг 9.14. Исходный код из файла jaas/JAASTest.policy

```

1 grant codebase "file:login.jar"
2 {
3     permission java.awt.AWTPermission
4         "showWindowWithoutWarningBanner";
5     permission java.awt.AWTPermission "accessEventQueue";
6     permission javax.security.auth.AuthPermission
7         "createLoginContext.Login1";
8     permission javax.security.auth.AuthPermission
9         "doAsPrivileged";
10    permission javax.security.auth.AuthPermission
11        "modifyPrincipals";
12    permission java.io.FilePermission
13        "jaas/password.txt", "read";
14 };
15
16 grant principal jaas.SimplePrincipal "role=admin"
17 {
18     permission java.util.PropertyPermission "*", "read";
19 };

```

Листинг 9.15. Исходный код из файла jaas/jaas.config

```

1 Login1
2 {
3     jaas.SimpleLoginModule required
4         pwfile="jaas/password.txt" debug=true;
5 };

```

javax.security.auth.callback.CallbackHandler 1.4

- **void handle(Callback[] callbacks)**

Обрабатывает указанные объекты обратного вызова, взаимодействуя при необходимости с пользователем, а затем сохраняет в них информацию, требующуюся для соблюдения безопасности.

javax.security.auth.callback.NameCallback 1.4

- **NameCallback(String prompt)**

- **NameCallback(String prompt, String defaultValue)**

Создают объект типа **NameCallback** с указанным приглашением и именем по умолчанию.

- **String getName()**

- **void setName(String name)**

Получают или устанавливают имя, приобретаемое с помощью текущего объекта обратного вызова.

- **String getPrompt()**

Получает приглашение, которое должно использоваться при запрашивании данного имени.

- **String getDefaultName()**

Получает имя по умолчанию, которое должно использоваться при запрашивании данного имени.

javax.security.auth.callback.PasswordCallback 1.4

- **PasswordCallback(String prompt, boolean echoOn)**

Создает объект типа **PasswordCallback** с указанным приглашением и признаком режима отображения эхо-символов.

- **char[] getPassword()**

- **void setPassword(char[] password)**

Получают или устанавливают пароль, получаемый с помощью текущего объекта обратного вызова.

- **String getPrompt()**

Получает приглашение, которое должно использоваться при запрашивании данного пароля.

- **boolean isEchoOn()**

Получает признак режима отображения эхо-символов, который должен использоваться при запрашивании данного пароля.

javax.security.auth.spi.LoginModule 1.4

- **void initialize(Subject subject, CallbackHandler handler, Map<String,?> sharedState, Map<String,?> options)**

Инициализирует объект типа **LoginModule** для аутентификации указанного субъекта. Для получения учетных данных использует указанный обработчик. Отображение **sharedState** применяется для взаимодействия с другими модулями регистрации, а отображение **options** — для хранения имен и значений, указанных при настройке данного экземпляра модуля регистрации.

javax.security.auth.spi.LoginModule 1.4 (окончание)

- **boolean login()**
Обеспечивает выполнение процесса регистрации и заполняет принципалы субъекта. Возвращает логическое значение `true`, если регистрация прошла успешно.
- **boolean commit()**
Если сценарий регистрации предполагает завершение в два этапа, то данный метод вызывается после успешного завершения всех модулей регистрации. Возвращает логическое значение `true`, если операция выполнена успешно.
- **boolean abort()**
Вызывается, если неудачный исход выполнения другого модуля предполагает отказ от регистрации. Возвращает логическое значение `true`, если операция выполнена успешно.
- **boolean logout()**
Отменяет регистрацию субъекта. Возвращает логическое значение `true`, если операция выполнена успешно.

9.4. Цифровые подписи

Как упоминалось ранее, интерес к платформе Java зародился с аплетов. Программисты сразу поняли, что с практической точки зрения, несмотря на широкие анимационные возможности, аплеты не полностью поддерживают модель безопасности. В версии JDK 1.0 аплеты очень строго контролировались. С другой стороны, в защищенной корпоративной сети компании риск атаки через аплеты минимален, поэтому логично было бы предоставить аплетам, выполняющимся в такой сети, дополнительные права. Разработчики из компании Sun Microsystems быстро осознали преимущества, которые дает применение аплетов, и поэтому решили предоставить пользователям возможность присваивать аплетам *разные* уровни защиты в зависимости от их происхождения. Так, если аплет поступает от надежного, заслуживающего доверия поставщика, то ему можно предоставить более обширные права доступа.

Для предоставления аплету дополнительных полномочий необходимо знать ответы на следующие два вопроса.

1. Откуда поступил аплет?
2. Не был ли его код поврежден во время передачи?

За последние 50 лет специалисты в области математики и информатики разработали немало сложных алгоритмов поддержки целостности данных и цифровых подписей. Многие из них реализованы в пакете `java.security`. Для применения таких алгоритмов совсем не обязательно понимать математические принципы, положенные в их основу. В последующих разделах описывается механизм свертки сообщений, позволяющий обнаружить факт изменения в документе, а также показывается, каким образом цифровая подпись способна подтверждать личность подписавшегося.

9.4.1. Свертки сообщений

Свертка сообщения — это цифровой “отпечаток” блока данных. Например, алгоритм безопасного хеширования SHA-1 (Secure Hash Algorithm #1) уплотняет любой блок данных в последовательность из 160 бит (20 байт). По аналогии с отпечатками пальцев, считается, что не существует двух одинаковых цифровых отпечатков по алгоритму SHA-1. На самом деле это не так, поскольку алгоритм SHA-1 поддерживает только 2^{160} отпечатков. Следовательно, теоретически они могут совпасть. Но число 2^{160} настолько велико, что вероятность дублирования цифровых отпечатков очень мала, но насколько? Как утверждается в книге *True Odds: How Risks Affect Your Everyday Life* Джеймса Уолша (James Walsh; издательство Merritt Publishing, 1996 г.), вероятность смерти от удара молнии составляет 1/30000. Если подсчитать вероятность такого же исхода для 10 человек (выбранных, например, злобным врагом), она окажется гораздо выше, чем вероятность наличия двух одинаковых цифровых отпечатков по алгоритму SHA-1. (Конечно, от удара молнии погибло гораздо больше, чем 10 человек, но здесь речь идет о специально выбранной группе людей.)

Свертка сообщения обладает двумя важными свойствами.

- Если изменяется один или несколько битов данных, то изменяется и свертка сообщения.
- Исходное сообщение нельзя изменить таким образом, чтобы полученное поддельное сообщение имело такую же свертку, как у исходного сообщения.

Второе свойство, конечно, соблюдается с определенной степенью вероятности. Допустим, что некий миллиардер составил следующее завещание.

“После смерти мое имущество должно быть разделено между моими детьми, но мой сын Джордж ничего не получит”.

Отпечаток данного сообщения по алгоритму SHA-1 имеет такой вид:

12 5F 09 03 E7 31 30 19 2E A6 E7 E4 90 43 84 B4 38 99 8F 67

Допустим, что недоверчивый миллиардер отдал текст завещания одному адвокату, а его отпечаток — другому. Допустим далее, что его сын Джордж подкупил адвоката, у которого хранится завещание, чтобы заменить слово *George* на слово *Bill*. В этом случае цифровой отпечаток по алгоритму SHA-1 поддельного завещания будет отличаться от аналогичного отпечатка исходного завещания:

7D F6 AB 08 EB 40 EC CD AB 74 ED E9 86 F9 ED 99 D1 45 B1 57

Может ли Джордж составить такое поддельное завещание, которое имело бы отпечаток оригинального завещания? Не может, потому что для перебора всех вариантов ему не хватило бы времени существования Земли, даже если бы он был счастливым обладателем миллиарда компьютеров, которые способны перебирать миллион вариантов завещания в секунду.

Для вычисления таких сверток сообщений был разработан целый ряд алгоритмов. Двумя наиболее известными из них являются: алгоритм SHA-1, разработанный в США Национальным институтом стандартов и технологий (National Institute of Standards and Technology — NIST), и алгоритм MD5, изобретенный

Рональдом Ривестом (Ronald Rivest) из Массачусетского технологического института (Massachusetts Institute of Technologies – MIT). Оба эти алгоритма способны шифровать фрагменты сообщений своим оригинальным способом. Подробнее с ними можно ознакомиться, например, в книге *Cryptography and Network Security, 5th Edition* Вильяма Столлингса (William Stallings; издательство Prentice Hall, 2011 г.). Следует, однако, иметь в виду, что недавно в обоих алгоритмах были обнаружены незначительные изъяны, и поэтому специалисты по шифрованию из института NIST рекомендуют пользоваться более надежными алгоритмами, в том числе SHA-256, SHA-384 или SHA-512.

В языке программирования Java реализованы алгоритмы MD5, SHA-1, SHA-256, SHA-384 и SHA-512. В частности, класс `MessageDigest` представляет собой *фабрику* для создания объектов, инкапсулирующих алгоритмы получения цифровых отпечатков. Этот класс содержит статический метод `getInstance()`, возвращающий экземпляр подкласса, производного от класса `MessageDigest`. Поэтому класс `MessageDigest` может выступать в роли фабричного класса или суперкласса для всех алгоритмов получения свертки сообщения. В приведенном ниже примере показано, каким образом получается объект для вычисления цифровых отпечатков по алгоритму SHA-1. А для того чтобы получить объект для вычисления цифровых отпечатков по алгоритму MD5, в качестве аргумента методу `getInstance()` следует предоставить строковое значение "MD5".

```
MessageDigest alg = MessageDigest.getInstance("SHA-1");
```

После создания объекта типа `MessageDigest` ему нужно передать все байты сообщения, повторно вызывая метод `update()`. Так, в приведенном ниже фрагменте кода содержимое файла передается полученному выше объекту `alg`, формирующему цифровой отпечаток.

```
InputStream in = . . .
int ch;
while ((ch = in.read()) != -1)
    alg.update((byte) ch);
```

С другой стороны, если байты размещаются в массиве, метод `update()` можно применить ко всему массиву следующим образом:

```
byte[] bytes = . . .;
alg.update(bytes);
```

Далее следует вызвать метод `digest()`, который дополняет вводимые данные недостающими битами (это необходимое условие для алгоритма получения цифровых отпечатков), вычисляет цифровой отпечаток и возвращает свертку сообщения в виде следующего байтового массива:

```
byte[] hash = alg.digest();
```

В примере программы из листинга 9.16 свертка сообщения вычисляется по алгоритму MD5, SHA-1, SHA-256, SHA-384 или SHA-512. Чтобы запустить ее на выполнение, введите одну из следующих команд:

```
java hash.Digest hash/input.txt
```

или

```
java hash.Digest hash/input.txt MD5
```

Листинг 9.16. Исходный код из файла hash/Digest.java

```

1 package hash;
2
3 import java.io.*;
4 import java.nio.file.*;
5 import java.security.*;
6
7 /**
8  * В этой программе вычисляется свертка сообщения из файла
9  * @version 1.20 2012-06-16
10 * @author Cay Horstmann
11 */
12 public class Digest
13 {
14     /**
15      * @param args args[0] - имя файла, args[1] - дополнительно
16      * алгоритм (SHA-1, SHA-256 или MD5)
17     */
18     public static void main(String[] args)
19         throws IOException, GeneralSecurityException
20     {
21         String algname = args.length >= 2 ? args[1] : "SHA-1";
22         MessageDigest alg = MessageDigest.getInstance(algname);
23         byte[] input = Files.readAllBytes(Paths.get(args[0]));
24         byte[] hash = alg.digest(input);
25         String d = "";
26         for (int i = 0; i < hash.length; i++)
27         {
28             int v = hash[i] & 0xFF;
29             if (v < 16) d += "0";
30             d += Integer.toString(v, 16).toUpperCase() + " ";
31         }
32         System.out.println(d);
33     }
34 }
```

java.security.MessageDigest 1.1

- **static MessageDigest getInstance(String algorithmName)**

Возвращает объект типа **MessageDigest**, реализующий указанный алгоритм. Если такой алгоритм не поддерживается, генерирует исключение типа **NoSuchAlgorithmException**.

- **void update(byte input)**

- **void update(byte[] input)**

- **void update(byte[] input, int offset, int len)**

Обновляют свертку сообщения, используя указанные байты.

- **byte[] digest()**

Вычисляет свертку сообщения по алгоритму хеширования, возвращает вычисленную свертку и устанавливает объект алгоритма в исходное состояние.

- **void reset()**

Устанавливает объект свертки сообщения в исходное состояние.

9.4.2. Подписание сообщений

В предыдущем разделе было показано, как создавать свертку сообщения, т.е. делать своего рода дактилоскопический отпечаток исходного сообщения. При изменении сообщения цифровой отпечаток измененного сообщения не будет совпадать с отпечатком исходного сообщения. Это означает, что при доставке сообщения вместе с его цифровым отпечатком получатель сможет проверить, не было ли оно подделано. Но если злоумышленнику удастся перехватить как сообщение, так и его исходный отпечаток, то он сможет легко изменить сообщение и переделать этот отпечаток так, как ему нужно. В конце концов, алгоритмы получения сверток сообщений всем известны и не требуют использования секретных ключей. В таком случае получатель поддельного сообщения и переделанного цифрового отпечатка так и не узнает, что его сообщение было изменено. Этот недостаток позволяют устраниТЬ цифровые подписи.

Чтобы стал понятнее принцип действия цифровых подписей, придется сначала ввести некоторые понятия из области шифрования *открытым ключом*. Основными в этой области являются такие понятия, как *открытый ключ* и *секретный ключ*. Открытый ключ сообщается всем, а секретный держится в строгом секрете. Соответствие между этими ключами устанавливается на основании математических отношений, которые здесь не рассматриваются. (Тем, кому интересно узнать о них более подробно, рекомендуется книга *The Handbook of Applied Cryptography*, оперативно доступная для заказа по адресу <http://www.cacr.math.uwaterloo.ca/hac/>.)

Эти ключи довольно длинные и сложные. В качестве примера ниже приведены пары открытого и секретного ключей, полученных с помощью алгоритма DSA (*Digital Signature Algorithm* — алгоритм создания цифровых подписей).

Открытый ключ:

p: fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617a
e01f35b91a47e6df63413c5e12ed0899bcd132acd50d99151bdc43ee737592e17

q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5

g: 678471b27a9cf44ee91a49c5147db1a9aaF244f05a434d6486931d2d14271b9
e35030b71fd73da179069b32e2935630e1c2062354d0da20a6c416e50be794ca4

y: c0b6e67b4ac098eb1a32c5f8c4c1f0e7e6fb9d832532e27d0bdab9ca2d2a8123
ce5a8018b8161a760480fadd040b927281ddb22cb9bc4df596d7de4d1b977d50

Соответствующий ему секретный ключ:

p: fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617a
e01f35b91a47e6df63413c5e12ed0899bcd132acd50d99151bdc43ee737592e17

q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5

g: 678471b27a9cf44ee91a49c5147db1a9aaF244f05a434d6486931d2d14271b9
e35030b71fd73da179069b32e2935630e1c2062354d0da20a6c416e50be794ca4

x: 146c09f881656cc6c51f27ea6c3a91b85ed1d70a

Считается, что вывести один ключ из другого практически невозможно. Несмотря на то что открытый ключ известен всем, узнать с его помощью секретный ключ злоумышленникам никогда не удастся, сколько бы вычислительных ресурсов ни было в их распоряжении.

В то, что никому не удастся вычислить секретный ключ на основе открытого ключа, трудно поверить, но, по крайней мере, пока еще никто не изобрел алгоритм, способный делать нечто подобное с ключами, генерируемыми с помощью самых распространенных в настоящее время алгоритмов шифрования. Из-за того что эти ключи достаточно длинные, для их расшифровки методом "грубой силы", т.е. путем простого перебора всех возможных вариантов, потребуется больше компьютеров, чем может быть вообще создано из всех атомов солнечной системы, и не одна тысяча лет. Разумеется, не исключено, что кому-нибудь удастся придумать для разгадывания ключей шифрования более эффективный алгоритм, чем простой перебор возможных вариантов.

Рассмотрим в качестве примера алгоритм шифрования RSA, изобретенный Ривестом (Rivest), Шамиром (Shamir) и Адлеманом (Adleman). Принцип действия этого алгоритма основывается на сложности разложения больших чисел на простые множители. За последние двадцать лет многие известные математики пытались разработать удачные алгоритмы разложения чисел на простые множители, но добиться этого пока еще никому не удалось. Из-за этого большинство специалистов по шифрованию считают, что в настоящее время ключи длиной 2000 битов абсолютно защищены от любых попыток взлома. Алгоритм шифрования DSA также считается довольно надежным. На рис. 9.11 показано, как выглядит процесс шифрования по алгоритму DSA на практике.

Допустим, что Алиса хочет послать сообщение Бобу, а Боб желает быть уверенным в том, что сообщение поступило именно от Алисы, а не от какого-то самозванца. Для этого Алиса составляет сообщение и *подписывает* свертку этого сообщения с помощью своего секретного ключа. Далее Боб получает копию ее открытого ключа и применяет его для *верификации* подписи. При удачном завершении процесса верификации Боб может быть уверен в следующем:

- исходное сообщение не было изменено;
- сообщение было подписано Алисой, обладающей секретным ключом, соответствующим тому открытому ключу, который Боб использовал для верификации.

Из этого примера ясно видно, почему так важны секретные ключи. Если кто-нибудь выкрадет секретный ключ Алисы или если правительство попросит ее расшифровать его, ей грозят неприятности, потому что выкравший ключ злоумышленник или представитель правительства сможет легко подделывать ее сообщения, заявки на пересылку денежных средств и выполнять другие подобные операции, а все будут считать, что все это делает сама Алиса.

9.4.3. Верификация подписи

В состав комплекта JDK входит утилита `keytool` командной строки, предназначенная для генерирования сертификатов и управления ими. Пока что она способна работать только в режиме командной строки, но можно надеяться, что в последующих выпусках JDK ее функциональные возможности будут доступны и в виде других, более удобных для пользователей версиях. В этом разделе она используется для того, чтобы показать, каким образом Алиса может подписывать документ и отправлять его Бобу, а Боб — проверять и удостоверяться в том, что документ был действительно подписан Алисой, а не каким-нибудь самозванцем.

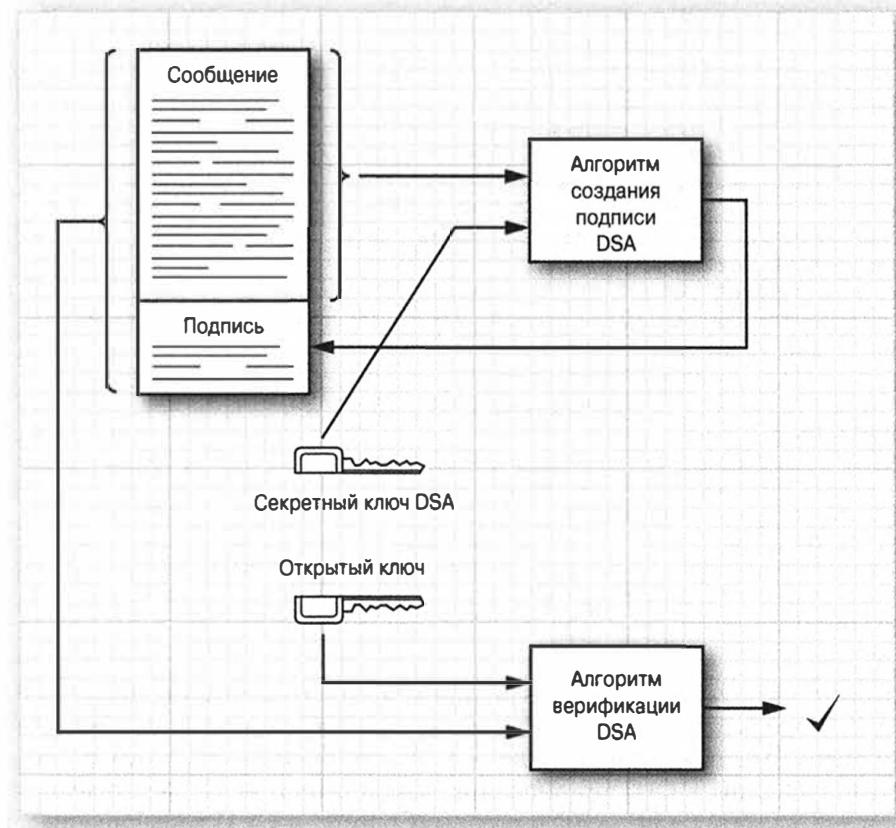


Рис. 9.11. Обмен сообщениями с использованием цифровой подписи, открытых ключей и алгоритма шифрования DSA

Утилита `keytool` позволяет управлять хранилищами ключей, базами данных сертификатов и парами секретных и открытых ключей. У каждой записи в хранилище ключей имеется свой псевдоним. Ниже показано, каким образом Алиса может создать хранилище `alice.store` и сгенерировать пару ключей с псевдонимом `alice`, воспользовавшись утилитой `keytool`.

```
keytool -genkeypair -keystore alice.certs -alias alice
```

При создании или открытии хранилища появляется приглашение ввести пароль. В данном примере вводится простое слово `secret`. Но при создании хранилища ключей с помощью утилиты `keytool` для каких-нибудь более серьезных целей рекомендуется выбрать надежный пароль и хранить его в полном секрете. При генерировании ключа появляется приглашение ввести следующую информацию:

```
Enter keystore password: secret
(Введите пароль для хранилища ключей)
```

```
Reenter new password: secret
(Еще раз введите новый пароль)
What is your first and last name?
(Ваше имя и фамилия [неизвестно])
[Unknown]: Alice Lee
What is the name of your organizational unit?
(Название подразделения вашей организации [неизвестно])
[Unknown]: Engineering Department
What is the name of your organization?
(Название вашей организации [неизвестно])
[Unknown]: ACME Software
What is the name of your City or Locality?
(Название вашего города или местности [неизвестно])
[Unknown]: San Francisco
What is the name of your State or Province?
(Название вашего штата или провинции [неизвестно])
[Unknown]: CA
What is the two-letter country code for this unit?
(Введите двухбуквенный код страны для данного
подразделения [неизвестно])
[Unknown]: US
Is <CN=Alice Lee, OU=Engineering Department, O=ACME Software,
L=San Francisco, ST=CA, C=US> correct?
(Все правильно?)
[no]: yes
```

В утилите keytool для идентификации владельцев ключей и создателей сертификатов используются имена по стандарту X.500 с составляющими для указания общего имени (Common Name – CN), организационного подразделения (Organizational Unit – OU), организации (Organization – O), местонахождения (Location – L), штата (State – ST) и страны (Country – C). И в завершение требуется указать пароль для ключа или нажать клавишу <Enter>, чтобы использовать в его качестве пароль хранилища ключей.

Допустим, что Алисе требуется предоставить Бобу копию своего открытого ключа. В таком случае ей нужно сначала экспорттировать файл сертификата следующим образом:

```
keytool -exportcert -keystore alice.certs -alias alice -file alice.cer
```

После этого она может отправить этот сертификат Бобу. Боб же, получив этот сертификат, может распечатать его по следующей команде:

```
keytool -printcert -file alice.cer
```

Ниже приведен пример такой распечатки. Если Боб захочет проверить, тот ли сертификат он получил, он может позвонить Алисе и сверить свою копию с исходным цифровым отпечатком сертификата по телефону.

```
Owner: CN=Alice Lee, OU=Engineering Department, O=ACME Software,
       L=San Francisco, ST=CA, C=US
Issuer: CN=Alice Lee, OU=Engineering Department, O=ACME Software,
       L=San Francisco, ST=CA, C=US
Serial number: 470835ce
Valid from: Sat Oct 06 18:26:38 PDT 2007 until:
               Fri Jan 04 17:26:38 PST 2008
```

Certificate fingerprints:

MD5: BC:18:15:27:85:69:48:B1:5A:C3:0B:1C:C6:11:B7:81
SHA1: 31:0A:A0:B8:C2:8B:3B:B6:85:7C:EF:C0:57:E5:94:95:61:47:6D:34
Signature algorithm name: SHA1withDSA
Version: 3



На заметку! Некоторые создатели сертификатов публикуют цифровые отпечатки своих сертификатов на веб-сайтах. Например, чтобы проверить сертификат компании VeriSign из каталога хранилища ключей `jre/lib/security/cacerts`, можно сначала отобразить содержимое этого каталога по следующей команде с параметром `-list`:

```
keytool -list -v -keystore jre/lib/security/cacerts
```

Паролем для доступа к данному хранилищу является слово `changeit`. А один из сертификатов в нем будет выглядеть так:

```
Owner: OU=VeriSign Trust Network, OU="(c) 1998 VeriSign, Inc. - For authorized use only", OU=Class 1 Public Primary Certification Authority - G2, O="VeriSign, Inc.", C=US
Issuer: OU=VeriSign Trust Network, OU="(c) 1998 VeriSign, Inc. - For authorized use only", OU=Class 1 Public Primary Certification Authority - G2, O="VeriSign, Inc.", C=US
Serial number: 4cc7eaaa983e71d39310f83d3a899192
Valid from: Sun May 17 17:00:00 PDT 1998 until:
              Tue Aug 01 16:59:59 PDT 2028
Certificate fingerprints:
  MD5: DB:23:3D:F9:69:FA:4B:B9:95:80:44:73:5E:7D:41:83
  SHA1: 27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32 :E5:47
```

Чтобы удостовериться в том, что данный сертификат является действительным, достаточно посетить веб-сайт компании VeriSign по адресу <http://www.verisign.com/repository/root.html>.

Убедившись в правильности сертификата, Боб может далее импортировать его в свое хранилище ключей по следующей команде:

```
keytool -importcert -keystore bob.certs -alias alice -file alice.cer
```



Внимание! Ни в коем случае не следует импортировать непроверенный сертификат в хранилище ключей. Ведь после ввода сертификата в хранилище ключей любая программа, пользующаяся хранилищем ключей, предполагает, что данный сертификат можно использовать для проверки цифровых подписей.

После этого Алиса может приступить к отправке Бобу подписанных документов. Подписывать и проверять архивные JAR-файлы можно с помощью утилиты `jarsigner`. Сначала Алисе следует ввести подписываемый документ в архивный JAR-файл:

```
jar cvf document.jar document.txt
```

Далее с помощью утилиты `jarsigner` она может добавить к этому файлу свою цифровую подпись. Для этого ей достаточно указать хранилище ключей, архивный JAR-файл и псевдоним используемого ключа в следующей команде:

```
jarsigner -keystore alice.certs document.jar alice
```

Получив этот файл, Боб может использовать для его верификации утилиту jarsigner с параметром **-verify**:

```
jarsigner -verify -keystore bob.certs document.jar
```

Указывать псевдоним ключа при этом Бобу не нужно. Утилита jarsigner сама отыщет в цифровой подписи указанное в формате X.500 имя владельца ключа и произведет поиск соответствующих ему сертификатов в хранилище ключей.

Если архивный JAR-файл не поврежден и цифровая подпись совпадает, то после этого утилиты jarsigner отобразит приведенное ниже сообщение. В противном случае вместо него появится сообщение об ошибке:

```
jar verified.
```

9.4.4. Проблема аутентификации

Допустим, вы получаете сообщение от своей подруги Алисы, которая подписала его описанным выше способом, используя свой секретный ключ. У вас уже может иметься копия ее открытого ключа, а иначе вам нетрудно будет получить ее у самой Алисы или на ее веб-странице. Наличие этого ключа позволяет легко удостовериться, что данное сообщение действительно написала Алиса и что оно не было кем-то подделано. А теперь допустим, что вы получаете сообщение от какого-то незнакомца, который представляется сотрудником известной компании по разработке программного обеспечения и предлагает вам воспользоваться прикрепленной им к данному сообщению программой. Этот незнакомец даже присыпает вам копию своего открытого ключа, чтобы вы могли удостовериться, что именно он является автором данного сообщения. Вы проверяете его и убеждаетесь, что подпись действительна. Это подтверждает лишь то, что сообщение было подписано с помощью соответствующего секретного ключа и что оно не было подделано.

Но имейте в виду, что *о самом авторе сообщения вам все равно ничего не известно!* Сгенерировать секретный и открытый ключи, подписать сообщение с помощью секретного ключа и отправить его вместе с подходящим открытым ключом мог кто угодно. Выяснение личности отправителя называется *проблемой аутентификации*.

Обычно проблема аутентификации разрешается очень просто. Например, у вас и у отправителя может быть общий знакомый, которому вы оба доверяете. Тогда отправитель может встретиться с вашим знакомым лично и передать ему диск с открытым ключом. Дальше ваш знакомый может встретиться с вами, подтвердить, что он действительно знает отправителя и что тот действительно работает на известную компанию по разработке программного обеспечения, после чего передать вам диск с открытым ключом (рис. 9.12). В таком случае получается, что гарантом аутентичности отправителя будет ваш знакомый.

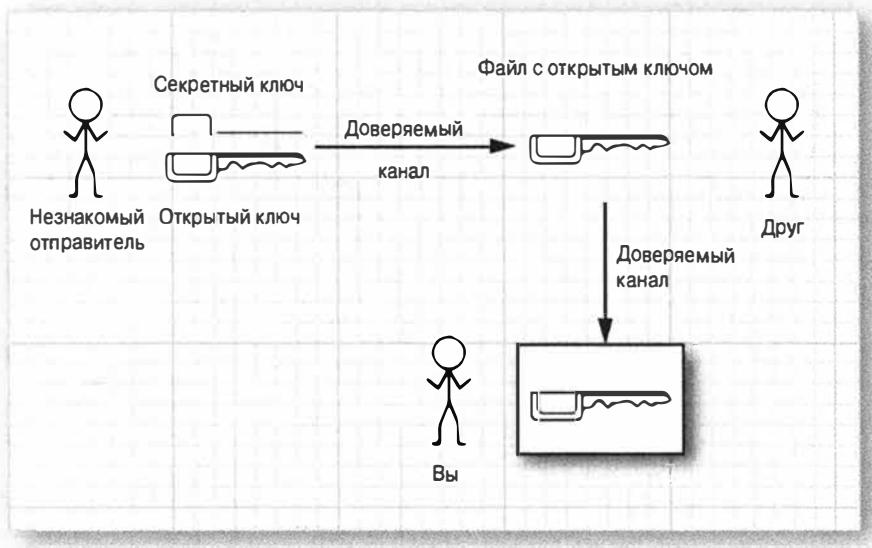


Рис. 9.12. Аутентификация через доверенного посредника

На самом деле вашему знакомому совсем не обязательно встречаться с вами. Вместо этого он может просто подписать файл с открытым ключом незнакомца с помощью своего секретного ключа и послать его вам (рис. 9.13). При получении файла с открытым ключом вы можете проверить подпись своего друга. А поскольку вы ему доверяете, то у вас нет сомнений, что он проверил учетные данные незнакомца, прежде чем добавлять свою цифровую подпись.

Но у вас может и не быть такого общего знакомого. В некоторых моделях доверительных отношений предполагается, что всегда существует некая "цепочка доверия", т.е. цепочка общих знакомых, всем звеням которой можно доверять. Но на практике так бывает далеко не всегда. Вы можете доверять своей подруге Алисе и знать, что она доверяет Бобу, но сами-то вы Боба не знаете, а следовательно, не до конца уверены, можно ли ему доверять. Другая модель доверительных отношений предполагает наличие какого-то одного хорошо зарекомендовавшего себя человека или организации, которой все могут безоговорочно доверять. Наиболее известным примером такой организации служит компания Verisign, Inc. (www.verisign.com).

Вам часто будут встречаться цифровые подписи, заверенные одной или несколькими ручающимися за аутентичность организациями, и поэтому придется самостоятельно оценивать, насколько им стоит доверять. Вы можете доверять компании VeriSign, например, потому, что часто встречаете ее логотип на многих веб-страницах, или потому, что слышали, будто бы в ней применяются невероятные меры безопасности.

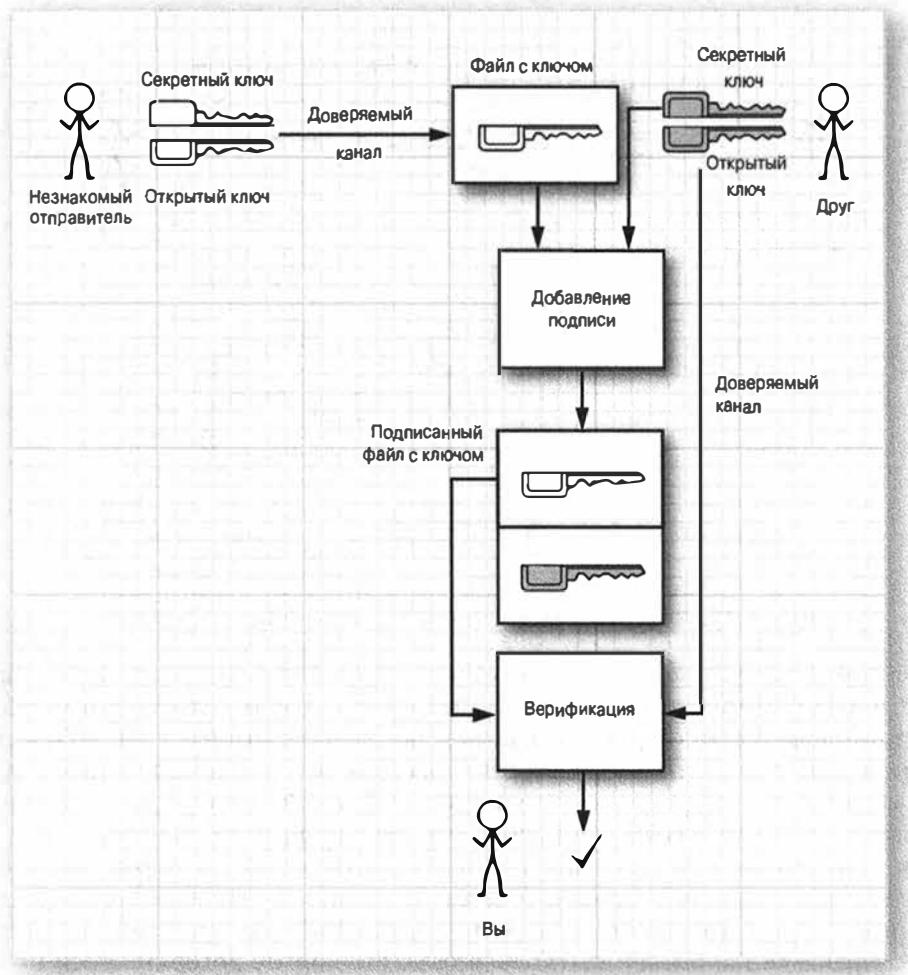


Рис. 9.13. Аутентификация через подпись доверенного посредника

Но вы все равно должны иметь ясное представление о том, каким образом в подобных организациях производится аутентификация и что именно они проверяют. Что касается компании VeriSign, то ее исполнительный директор, естественно, не встречается лично с каждым человеком или представителем компании при аутентификации ключа. Получить идентификатор "класса 1" от этой компании можно, заполнив веб-форму и заплатив небольшой взнос. В этой форме нужно указать имя, организацию, страну и адрес электронной почты. После заполнения формы по указанному адресу отсылается ключ или инструкции, как его получить. Таким образом, уверенным можно быть только в подлинности электронного адреса, потому что имя и название организации не проверяются и могут быть любыми. Существуют и более строгие классы идентификаторов, например, при желании

получить идентификатор “класса 3” компания VeriSign потребует представить отчет о финансовом положении, официально заверенный у нотариуса. У других организаций, занимающихся аутентификацией, будут другие требования. Поэтому очень важно, чтобы при получении заверенного цифровой подписью сообщения вы ясно понимали, каким образом проверялась его подлинность.

9.4.5. Подписание сертификатов

В разделе 9.4.3 было показано, как Алиса использовала самостоятельно подписываемый сертификат для передачи открытого ключа Бобу. Но Бобу пришлось проверить подлинность этого сертификата путем его сверки с исходным цифровым отпечатком от Алисы.

Допустим теперь, что Алисе требуется послать подписанное сообщение своей коллеге Синди, а та не любит тратить время на сверку цифровых подписей с исходными отпечатками. В этом случае Синди может помочь посредническая организация, которой она могла бы доверить проверку всех цифровых подписей. В рассматриваемом здесь примере предполагается, что такой организацией является отдел информационных ресурсов компании ACME Software.

Этот отдел занимается выдачей сертификатов, т.е. представляет собой своего рода *центр сертификации*. У каждого сотрудника компании ACME Software имеется полученный из этого отдела открытый ключ, который находится в хранилище ключей и перед установкой был тщательно сверен с цифровым отпечатком системным администратором. Этот центр сертификации занимается также подписанием ключей всех сотрудников компании ACME Software. Благодаря этому хранилище ключей автоматически доверяет устанавливаемым ключам сотрудников данной компании, поскольку они подписаны с помощью доверяемого ключа.

Ниже поясняется, как сымитировать этот процесс. Сначала создается хранилище ключей `acmesoft.certs`. Затем генерируется пара ключей и открытый ключ экспортируется по следующим командам:

```
keytool -genkeypair -keystore acmesoft.certs -alias acmeroot  
keytool -exportcert -keystore acmesoft.certs -alias acmeroot  
-file acmeroot.cer
```

Открытый ключ экспортируется в “самозаверенный” сертификат. Далее он вводится в хранилище ключей каждого сотрудника следующим образом:

```
keytool -importcert -keystore cindy.certs -alias acmeroot  
-file acmeroot.cer
```

Чтобы иметь возможность отправлять подписанные сообщения Синди и любым другим сотрудникам компании ACME Software, Алисе нужно принести свой сертификат в отдел информационных ресурсов данной компании и подписать его. Утилита `keytool`, к сожалению, не предоставляет никаких средств для решения подобной задачи. Для восполнения этого пробела в исходном коде, прилагаемом к данной книге, предоставляется класс `CertificateSigner`. В этом случае уполномоченному сотруднику компании ACME Software остается только проверить личность Алисы и сформировать подписанный сертификат по следующей команде:

```
java CertificateSigner -keystore acmesoft.certs -alias acmeroot  
-infile alice.cer -outfile alice_signedby_acmeroot.cer
```

Очевидно, что выполнение такой команды связано с известным риском и требует, чтобы у применяемой для подписания сертификата программы был доступ к хранилищу ключей компании ACME Software, а у уполномоченного сотрудника — необходимый для этого пароль.

После этого Алиса может передать файл `alice_signedby_acmeroott.cer` Синди и любому другому сотруднику ACME Software. В качестве альтернативного варианта этот файл может быть также сохранен в каталоге компании. Напомним, что в этом файле содержится открытый ключ Алисы и подтверждение компании ACME Software, что данный ключ действительно принадлежит Алисе. Далее Синди может импортировать подписанный сертификат Алисы в свое хранилище ключей по следующей команде:

```
keytool -importcert -keystore cindy.certs -alias alice -file  
alice_signedby_acmeroott.cer
```

При выполнении этой команды в хранилище ключей будет автоматически проверено, был ли данный сертификат подписан с помощью уже присутствующего в нем доверяемого корневого ключа. Благодаря этому Синди не придется сверять его с исходным цифровым отпечатком данного сертификата. Введя один раз корневой сертификат и сертификаты всех остальных лиц, часто присылающих ей документы, Синди избавляется от необходимости впредь беспокоиться о хранилище ключей.

9.4.6. Запросы сертификатов

В предыдущем разделе был рассмотрен пример имитации центра сертификации с хранилищем ключей и классом `CertificateSigner`. Но в большинстве центров сертификации для управления сертификатами применяется более сложное программное обеспечение и несколько иные форматы сертификатов. В этом разделе обсуждаются дополнительные действия, которые требуется выполнить для организации нормального взаимодействия с подобным программным обеспечением.

Выберем для примера пакет программного обеспечения OpenSSL. Это программное обеспечение, как правило, предварительно устанавливается во многих системах Linux и Mac OS X вместе с портом Cygwin. Если оно не установлено, его можно без особого труда загрузить по адресу <http://www.openssl.org>.

Чтобы создать центр сертификации, следует запустить сценарий CA. Точное месторасположение этого сценария зависит от конкретной операционной системы. Так, в ОС Ubuntu его можно запустить по следующей команде:

```
/usr/lib/ssl/misc/CA.pl -newca
```

Этот сценарий создаст в текущем каталоге подкаталог `demoCA`, содержащий пару корневых ключей и хранилище для сертификатов и списков аннулирования сертификатов.

Далее необходимо импортировать открытый ключ в хранилище ключей, созданное на Java для всех сотрудников. Но из-за того что этот ключ имеет формат PEM (Privacy Enhanced Mail — Почта повышенной секретности), а не легко распознаваемый хранилищем ключей формат DER, сначала придется скопировать файл `demoCA/cacert.pem` в файл `acmeroott.pem`, открыть его

в текстовом редакторе и удалить все, что находится перед строкой -----BEGIN CERTIFICATE----- и после строки -----END CERTIFICATE-----.

После этого файл acmeroot.pem можно скопировать в каждое хранилище ключей обычным способом, как показано ниже. Невероятно, но факт: утилита keytool не способна выполнить эту операцию редактирования самостоятельно.

```
keytool -importcert -keystore cindy.certs -alias alice  
-file acmeroot.pem
```

Чтобы подписать открытый ключ Алисы, нужно сначала сформировать *запрос сертификата* в формате PEM по следующей команде:

```
keytool -certreq -keystore alice.store -alias alice  
-file alice.pem
```

Далее для подписания этого сертификата нужно выполнить следующую команду:

```
openssl ca -in alice.pem -out alice_signedby_acmeroot.pem
```

После этого, как и прежде, из файла alice_signedby_acmeroot.pem следует удалить все строки, которые находятся за пределами маркеров BEGIN CERTIFICATE/END CERTIFICATE, и затем импортировать этот файл в хранилище ключей по приведенной ниже команде. Аналогичным образом подписать сертификат можно и в каком-нибудь другом общедоступном центре сертификации, подобном компании VeriSign.

```
keytool -importcert -keystore cindy.certs -alias alice  
-file alice_signedby_acmeroot.pem
```

9.4.7. Подписание кода

Технология аутентификации чаще всего применяется для подписания исполняемых программ. Копируя программу из сети, пользователь вполне обоснованно может потребовать гарантii ее подлинности, поскольку такая программа может нанести вред, если она заражена каким-нибудь вирусом. Поэтому требуется полная уверенность в том, что программа предоставлена надежным источником и во время пересылки не была изменена.

В этом разделе будет показано, каким образом подписываются архивные JAR-файлы и как настраиваются средства Java для проверки достоверности подписи. Такая возможность была предусмотрена для подключаемого модуля Java Plug-in, предназначенного для запуска аплетов и приложений Java Web Start. И хотя эти технологии не находят большого широкого применения, тем не менее, их, возможно, придется поддерживать в унаследованных программных продуктах.

В первоначальной версии Java аплеты допускалось выполнять в "песочнице" с ограниченными полномочиями непосредственно после их загрузки. Если же пользователям требовались аплеты, способные получать доступ к локальной файловой системе, устанавливать сетевые соединения и т.д., они должны были явным образом давать свое согласие на подобные операции. Чтобы гарантировать от намеренного повреждения кода аплета в процессе его выполнения, такой код приходилось снабжать цифровой подписью.

Допустим, что, выйдя в Интернет, пользователь открывает веб-страницу, на которой предлагается запустить аплет или приложение Web Start неизвестного поставщика, как, например, показано на рис. 9.14. Такой аплет подписан сертификатом разработчика программного обеспечения. В диалоговом окне указывается разработчик данной программы и создатель сертификата. А пользователь должен решить, следует лишь санкционировать выполнение выбранной прикладной программы.

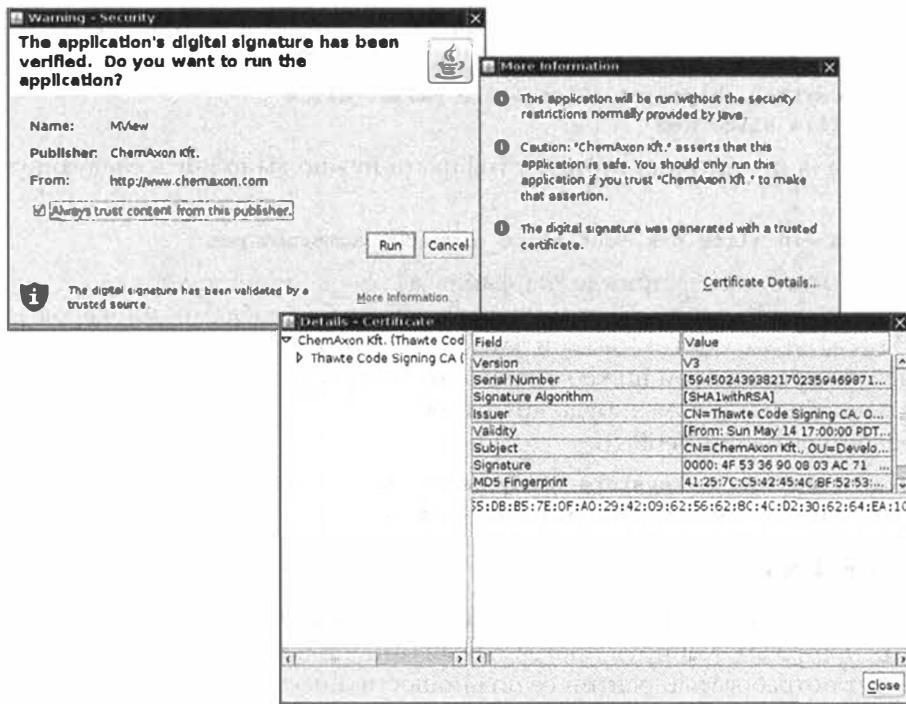


Рис. 9.14. Запуск подписанного аплета

Попробуем выяснить, что известно пользователю в подобной ситуации и что может повлиять на его решение?

- Компания Thawte продала сертификат разработчику программного обеспечения.
- Аплет действительно подписан этим сертификатом и во время пересылки не был изменен.
- Сертификат действительно подписан компанией Thawte и проверен с помощью открытого ключа, который находится в локальном файле cacerts.

Значит ли это, что данный код надежен? Следует ли доверять поставщику, если известно только его имя и тот факт, что компания Thawte продала ему сертификат разработчика программного обеспечения? Вряд ли компания Thawte сможет гарантировать, что под именем разработчика программного обеспечения ChemAxon Kft. не скрывается явный злоумышленник. Более того, создатели

сертификатов не претендуют на это, т.е. они не занимаются проверкой честности и компетентности поставщиков программного обеспечения. Они проверяют подлинность поставщиков программного обеспечения по сканированной копии их лицензии на ведение коммерческой деятельности или паспорта.

Как видите, такое решение нельзя считать удовлетворительным. Вероятно, было бы лучше расширить функциональные возможности "песочницы". Когда была впервые внедрена технология Java Web Start, она вышла за пределы "песочницы", разрешая пользователям давать согласие на ограниченный доступ к файлам или печатающему устройству. Но такой подход не получил дальнейшего развития. Напротив, в компании Oracle решили больше не поддерживать неподписанные аплеты из-за постоянных хакерских атак на них.

В настоящее время аплеты применяются редко и, главным образом, в прежних целях. Если же требуется поддержка общедоступного аплета, его следует подписать сертификатом такого поставщика, который заслуживает доверия в исполняемой системе Java.

Чуть лучше дело обстоит с приложениями для внутренних корпоративных сетей, где на локальных машинах можно установить файлы правил защиты и сертификаты, чтобы исключить взаимодействие с пользователями, требующееся для запуска на выполнение прикладного кода из доверяемых источников. Всякий раз, когда подключаемый модуль Java Plug-in загружает подписанный код, он обращается к файлу правил защиты за полномочиями, а к хранилищу ключей — за подписями.

Далее в этом разделе поясняется, каким образом создаются файлы правил защиты, дающие разрешение на выполнение прикладного кода из известных источников. Построение и развертывание подобных файлов — дело не конечных пользователей, а системных администраторов, в обязанности которых входит подготовка и распространение программ для внутренних корпоративных сетей.

Допустим, что компании ACME Software требуется, чтобы ее сотрудники могли запускать определенные программы, требующие доступа к локальным файлам, но сделать так, чтобы эти программы были доступны через браузер в виде аплетов или приложений Web Start. Как было показано ранее в этой главе, компания ACME Software могла бы идентифицировать программы по их кодовой базе. Но это означает, что ей пришлось бы обновлять файлы прав защиты всякий раз, когда программы переносятся на другой сервер. Поэтому в компании ACME Software вместо этого было принято решение подпisyывать архивные JAR-файлы, содержащие программный код.

Для этого сначала формируется корневой сертификат по следующей команде:

```
keytool -genkeypair -keystore acmesoft.certs -alias acmeroot
```

Разумеется, хранилище ключей, содержащее секретный корневой ключ, должно обязательно находиться в безопасном месте. Поэтому создается второе хранилище ключей client.certs для размещения в нем открытых сертификатов и в него сразу же вводится открытый сертификат acmeroot:

```
keytool -exportcert -keystore acmesoft.certs -alias acmeroot  
-file acmeroot.cer  
keytool -importcert -keystore client.certs -alias acmeroot  
-file acmeroot.cer
```

В процессе создания подписанного архивного JAR-файла программисты могут добавлять свои файлы классов в этот архивный JAR-файл обычным образом, как показано в приведенном ниже примере.

```
javac FileReadApplet.java
jar cvf FileReadApplet.jar *.class
```

Далее уполномоченный сотрудник компании ACME Software может запустить утилиту jarsigner, указав архивный JAR-файл и псевдоним секретного ключа следующим образом:

```
jarsigner -keystore acmsoft.certs FileReadApplet.jar acmeroot
```

После этого аплет можно считать готовым к развертыванию на веб-сервере. А теперь перейдем к настройке безопасности на стороне клиента. Файл правил защиты должен рассыпаться каждой клиентской машине. Для ссылки на хранилище ключей файл правил защиты начинается со следующей строки:

```
keystore "URL_хранилища_ключей", "тип_хранилища_ключей";
```

Указанный URL может быть как абсолютным, так и относительным. Относительные URL являются таковыми по отношению к месту расположения файла правил защиты. Если хранилище формировалось с помощью утилиты keytool, оно будет относиться к типу JKS, как показано в приведенном ниже примере.

```
keystore "client.certs", "JKS";
```

Далее, операторы grant в файле правил защиты могут быть снабжены суффиксами **signedBy "псевдоним"**, как выделено ниже полужирным. В этих операторах предоставляются полномочия любому коду, который может быть проведен с помощью открытого ключа, связанного с указанным псевдонимом.

```
grant signedBy "acmeroot"
{
    ...
};
```

Вы можете опробовать процесс подписания кода на примере аплета (листинг 9.17), который пытается прочитать локальный файл. Правила защиты по умолчанию разрешают ему считывать файлы только из каталога кодовой базы и любых его подкаталогов. Воспользуйтесь утилитой appletviewer для просмотра аплетов, чтобы запустить этот аплет и удостовериться, что с его помощью можно просматривать файлы только из каталога и подкаталогов кодовой базы и никаких других каталогов.

Для целей рассматриваемого здесь примера предоставляется файл правил защиты applets.policy, содержащий следующее:

```
keystore "client.certs", "JKS";
grant signedBy "acmeroot"
{
    permission java.lang.RuntimePermission "usePolicy";
    permission java.io.FilePermission "/etc/*", "read";
};
```

В этих правилах защиты полномочия usePolicy переопределены полномочия типа “все или ничего”, используемые по умолчанию для подписанных аплетов, а также указывается, что всем аплетам, которые подписаны с помощью

сертификата acmeroot, разрешается считывать файлы из каталога /etc (пользователям Windows этот каталог нужно заменить на какой-нибудь другой, например C:\Windows).

Теперь осталось лишь указать утилите appletviewer для просмотра аплетов, чтобы она использовала этот файл правил защиты. Ниже показано, как это делается. После этого аплет сможет считывать файлы из каталога /etc, демонстрируя тем самым принцип действия механизма, применяемого для подписания кода.

```
appletviewer -J-Djava.security.policy=applet.policy  
FileReadApplet.html
```



Совет! Если вы испытываете затруднения при выполнении упомянутого выше действия, введите дополнительный параметр **-J-Djava.security.debug-policy**, чтобы вывести подробные сообщения, позволяющие отслеживать порядок соблюдения программой установленных правил защиты.

В качестве заключительной проверки попробуйте запустить аплет в браузере (рис. 9.15). Для этого вам придется скопировать файл полномочий и хранилище ключей в каталог развертывания Java. В UNIX и Linux это каталог .java/deployment, находящийся в начальном каталоге, а в Windows — каталог C:\Users\регистрационное_имя_пользователя\AppData\Sun\Java\Deployment. Ради удобства он будет называться в дальнейшем *deploydir*, т.е. каталог развертывания программ.

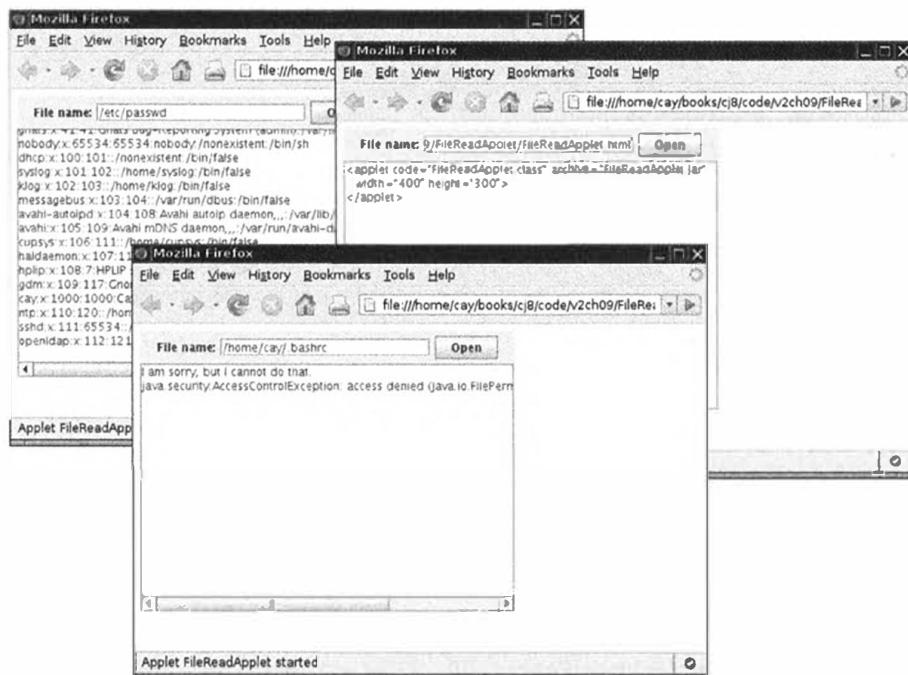


Рис. 9.15. Подписанный аплет имеет доступ к локальным файлам для чтения

Скопируйте файлы applet.policy и client.certs в каталог *deploydir/security*. Переименуйте в этом каталоге файл applets.policy на java.policy. (Сделайте это очень аккуратно, чтобы не удалить случайно существующий файл java.policy. Если таковой имеется, введите в него содержимое файла applet.policy.)

Перезапустите браузер и загрузите файл FileReadApplet.html. При этом не должно быть никакого приглашения на принятие сертификата. Далее удостоверьтесь, что файлы можно загружать из каталога /etc и каталога с аплетом, а из всех остальных каталогов — нельзя.

По завершении не забудьте очистить каталог *deploydir/security*, удалив из него файлы java.policy и client.certs. Затем перезапустите браузер. Если после очистки каталога вы снова попробуете загрузить аплет, то у вас уже не должно быть полномочий на чтение файлов из локальной файловой системы. Вместо этого на экране появится приглашение на принятие сертификата. Подробнее о сертификатах безопасности речь пойдет в следующем разделе.



Совет! Подробнее о настройке клиентских параметров безопасности на платформе Java можно узнать в документе *Java Rich Internet Applications Guide* [Руководство по насыщенным интернет-приложениям Java], доступном по адресу <http://docs.oracle.com/javase/8/docs/technotes/guides/jweb/index.html>.

Листинг 9.17. Исходный код из файла signed/FileReadApplet.java

```

1 package signed;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.io.*;
6 import java.nio.file.*;
7 import javax.swing.*;
8
9 /**
10 * Этот аплет можно выполнять вне "лесочницы", чтобы читать
11 * локальные файлы, как только он наделяется нужными полномочиями
12 * permissions.
13 * @version 1.13 2016-05-10
14 * @author Cay Horstmann
15 */
16 public class FileReadApplet extends JApplet
17 {
18     private JTextField fileNameField;
19     private JTextArea fileText;
20
21     public void init()
22     {
23         EventQueue.invokeLater(() ->
24         {
25             fileNameField = new JTextField(20);
26             JPanel panel = new JPanel();
27             panel.add(new JLabel("File name:"));
28             panel.add(fileNameField);
29             JButton openButton = new JButton("Open");
30             panel.add(openButton);

```

```
31     ActionListener listener = event ->
32         loadFile(fileNameField.getText());
33     fileNameField.addActionListener(listener);
34     openButton.addActionListener(listener);
35     add(panel, "North");
36     fileText = new JTextArea();
37     add(new JScrollPane(fileText), "Center");
38 };
39 }
40
41 /**
42 * Загружает содержимое файла в текстовую область
43 * @param filename the file name
44 */
45 public void loadFile(String filename)
46 {
47     fileText.setText("");
48     try
49     {
50         fileText.append(new String(Files.readAllBytes(
51             Paths.get(filename))));}
52     }
53     catch (IOException ex)
54     {
55         fileText.append(ex + "\n");
56     }
57     catch (SecurityException ex)
58     {
59         fileText.append("I am sorry, but I cannot do that.\n");
60         fileText.append(ex + "\n");
61         ex.printStackTrace();
62     }
63 }
64 }
```

9.5. Шифрование

До сих пор рассматривалась аутентификация с помощью цифровых подписей. Еще одним важным средством обеспечения безопасности является *шифрование*. Информация, заверенная цифровой подписью, доступна для просмотра, а подпись лишь подтверждает, что эта информация не была изменена. Для просмотра зашифрованных данных этого недостаточно, поскольку их нужно расшифровать с помощью согласованного ключа.

Аутентификации оказывается достаточно для подписания кода, который не нужно скрывать. А шифрование требуется в тех случаях, когда аплеты или приложения передают конфиденциальную информацию, например, номера кредитных карточек и прочие личные данные.

В прошлом во многих компаниях существовали патентные и экспортные ограничения на использование эффективных алгоритмов шифрования. Теперь же все эти ограничения стали, к счастью, менее жесткими, а срок патентных ограничений на использование ряда важных алгоритмов шифрования и вовсе истек. В стандартной библиотеке текущей версии Java SE предусмотрены достаточно хорошие средства шифрования.

9.5.1. Симметричные шифры

Криптографические расширения Java содержат класс `Cipher`, который является суперклассом для всех классов, имеющих отношение к шифрованию. Для создания объекта, реализующего алгоритм шифрования, используется метод `getInstance()` одним из следующих способов:

```
Cipher cipher = Cipher.getInstance(algorithmName);
```

или

```
Cipher cipher = Cipher.getInstance(algorithmName, providerName);
```

В комплекте JDK для всех шифров используется поставщик под названием SunJCE. Если имя поставщика не указано явно, то по умолчанию принимается имя SunJCE. Если же требуется воспользоваться алгоритмами шифрования, которые не поддерживаются инструментальными средствами компании Oracle, следует указать другого поставщика. Название алгоритма шифрования задается в виде символьной строки, например "DES" или "DES/CBC/PKCS5Padding".

Алгоритм DES (Data Encryption Standard — стандарт шифрования данных) — один из наиболее старых алгоритмов шифрования с длиной ключа 56 бит. В настоящее время он считается устаревшим, поскольку может взламываться методом "грубой силы" (соответствующий пример можно найти по адресу http://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker). Намного более эффективным является появившийся после него алгоритм AES (Advanced Encryption Standard — усовершенствованный стандарт шифрования), подробное описание которого можно найти по адресу www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf. В рассматриваемых далее примерах будет использоваться алгоритм шифрования AES.

Как только объект, реализующий алгоритм шифрования, будет создан, его нужно инициализировать, указав режим и ключ шифрования, т.е. установив параметры `mode` и `key` следующим образом:

```
int mode = . . .;
Key key = . . .;
cipher.init(mode, key);
```

Параметр `mode` может принимать значение одной из следующих констант:

```
Cipher.ENCRYPT_MODE
Cipher.DECRYPT_MODE
Cipher.WRAP_MODE
Cipher.UNWRAP_MODE
```

Режимы свертывания и развертывания применяются для шифрования одного ключа на основе другого. Пример применения таких режимов будет приведен в следующем разделе. После этого можно повторно вызывать метод `update()` для шифрования всех требующихся блоков данных, как показано ниже.

```
int blockSize = cipher.getBlockSize();
byte[] inBytes = new byte[blockSize];
. . . // прочитать байты из входного массива inBytes
int outputSize= cipher.getOutputSize(blockSize);
byte[] outBytes = new byte[outputSize];
int outLength = cipher.update(inBytes, 0, outputSize, outBytes);
. . . // записать байты в выходной массив outBytes
```

По завершении следует вызвать метод `doFinal()` один раз. Если доступен последний блок вводимых данных (меньшего объема, чем указано в переменной `blockSize`), то данный метод нужно вызывать следующим образом:

```
outBytes = cipher.doFinal(inBytes, 0, inLength);
```

А если были зашифрованы все вводимые данные, то данный метод следует вызвать так:

```
outBytes = cipher.doFinal();
```

Вызывать метод `doFinal()` требуется для того, чтобы заполнить завершающий блок данных. Рассмотрим, например, алгоритм шифрования DES, в котором размер блока составляет 8 байт. Допустим, что размер последнего блока вводимых данных меньше 8 байт. Разумеется, недостающие байты можно дополнить нулями таким образом, чтобы блок занимал 8 байт, а затем зашифровать его. Но после расшифровки блоков данных полученный результат будет содержать несколько присоединенных в конце нулей, а следовательно, он будет несколько отличаться от исходного файла данных. Это может представлять определенное затруднение, и для ее преодоления как раз и требуется *схема заполнения*. Одной из наиболее часто применяемых является схема заполнения, описанная в документе Public Key Cryptography Standard (PKCS — стандарт шифрования открытым ключом) #5 специалистами из компании RSA Security, Inc. (<ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf>).

В этой схеме последний блок заполняется не нулями, а числами, равными недостающему количеству байтов. Иными словами, если L — это последний (неполный) блок данных, то он будет заполнен следующим образом:

L 01	если <code>length(L) = 7</code>
L 02 02	если <code>length(L) = 6</code>
L 03 03 03	если <code>length(L) = 5</code>
...	
L 07 07 07 07 07 07 07	если <code>length(L) = 1</code>

И наконец, если длина вводимых данных делится нацело на 8, то к вводимым данным присоединяется и затем шифруется только один такой блок, как показано ниже. А при расшифровке на количество отбрасываемых заполняющих символов указывает самый последний байт простого текста.

08 08 08 08 08 08 08 08

9.5.2. Генерирование ключей шифрования

Для шифрования требуется сгенерировать ключ. Каждый алгоритм шифрования предусматривает свой формат для ключей, но самое главное, чтобы их генерирование выполнялось произвольным образом. Чтобы получить сгенерированный совершенно произвольно ключ, требуется выполнить три действия.

1. Создать объект типа `KeyGenerator`.
2. Инициализировать генератор случайных чисел. Если блок шифра имеет переменную длину, необходимо также указать желаемую длину блока.
3. Вызвать метод `generateKey()`.

В качестве примера ниже показано, каким образом генерируется ключ шифрования по алгоритму AES.

```
KeyGenerator keygen = KeyGenerator.getInstance("AES");
SecureRandom random = new SecureRandom(); // см. пояснение ниже
keygen.init(random);
Key key = keygen.generateKey();
```

С другой стороны, ключ шифрования можно также сгенерировать из фиксированного набора необработанных данных (например, пароля или вводимых с клавиатуры символов). В таком случае следует воспользоваться классом SecretKeyFactory:

```
byte[] keyData = . . .;
// 16 байт для ключа шифрования по алгоритму AES
SecretKey key = new SecretKeySpec(keyData, "AES");
```

При генерировании ключей следует удостовериться, что для этой цели используются *подлинно случайные* числа. Например, обычный генератор случайных чисел, реализуемый в классе Random, где в роли начальных значений выступают текущие дата и время, не будет генерировать случайные в достаточной степени числа. Допустим, что системные часы показывают время с точностью до 1/10 секунды. Это означает, что на каждый день может приходиться максимум по 864000 начальных значений. Следовательно, узнав день генерирования ключа шифрования (его зачастую нетрудно вычислить по дате сообщения или дате истечения срока действия сертификата), злоумышленнику останется лишь сгенерировать все возможные значения за этот день.

Класс SecureRandom генерирует намного более надежные случайные числа, чем класс Random. Но предоставлять ему начальное значение, с которого должна начинаться числовая последовательность в произвольной точке, все равно придется. Для этого удобнее всего получить случайные данные, вводимые из какого-нибудь аппаратного устройства вроде генератора белого шума. С другой стороны, случайные данные можно получить, запросив у пользователя ввести любую бессмысленную последовательность символов, из которой в качестве начального случайного значения должен выбираться только один или два бита. После накопления таких случайных битов в массив байтов они должны быть просто переданы методу setSeed(), как показано ниже.

```
SecureRandom secrand = new SecureRandom();
byte[] b = new byte[20];
// заполнить массив подлинно случайными битами
secrand.setSeed(b);
```

Если не указать генератору случайных чисел никакого начального значения, он сформирует собственное 20-байтовое значение, запустив несколько потоков на исполнение, переведя их в режим ожидания и рассчитав точное время их выхода из режима ожидания.



На заметку! Трудно сказать, является ли такой алгоритм безопасным. Раньше алгоритмы, действие которых основывалось на вычислении временных показателей некоторых компонентов компьютера, например времени доступа к жесткому диску, зачастую оказывались на поверку генерирующими недостаточно случайные числа.

В примере программы из листинга 9.18 демонстрируется применение алгоритма шифрования AES. А сервисный метод шифрования `crypt()` из листинга 9.19 будет еще не раз использован в остальных примерах далее в этой главе. Для того чтобы воспользоваться данной программой, нужно сначала сгенерировать секретный ключ, выполнив следующую команду:

```
java aes.AESTest -genkey secret.key
```

Полученный секретный ключ следует сохранить в файле `secret.key`. После этого шифрование можно выполнить по следующей команде:

```
java aes.AESTest -encrypt plaintextFile encryptedFile secret.key
```

А расшифровывание производится по такой команде:

```
java aes.AESTest -decrypt encryptedFile decryptedFile secret.key
```

Рассматриваемая здесь программа довольно проста. Сначала с помощью параметра `-genkey` генерируется новый секретный ключ, который затем сохраняется в указанном файле. Эта операция обычно занимает много времени, поскольку большая ее часть уходит на инициализацию генератора случайных чисел. Параметры `-encrypt` и `-decrypt` предусматривают вызов одного и того же метода `crypt()`, который, в свою очередь, вызывает такие методы для объекта шифрования, как `update()` и `doFinal()`. Следует, однако, иметь в виду, что метод `update()` вызывается повторно до тех пор, пока блоки вводимых данных не достигнут полной длины, и что метод `doFinal()` вызывается с частично заполненным блоком вводимых данных (который затем заполняется) или вообще безо всяких дополнительных данных (для формирования одного заполняющего блока).

Листинг 9.18. Исходный код из файла `aes/AESTest.java`

```
1 package aes;
2
3 import java.io.*;
4 import java.security.*;
5 import javax.crypto.*;
6 /**
7  * В этой программе проверяется шифрование по алгоритму AES.
8  * Применение:
9  *   java aes.AESTest -genkey keyfile
10 *   java aes.AESTest -encrypt plaintext encrypted keyfile
11 *   java aes.AESTest -decrypt encrypted decrypted keyfile
12 * @author Cay Horstmann
13 * @version 1.01 2012-06-10
14 */
15 public class AESTest
16 {
17     public static void main(String[] args) throws IOException,
18                         GeneralSecurityException, ClassNotFoundException
19     {
20         if (args[0].equals("-genkey"))
21         {
22             KeyGenerator keygen = KeyGenerator.getInstance("AES");
23             SecureRandom random = new SecureRandom();
24             keygen.init(random);
```

```

25     SecretKey key = keygen.generateKey();
26     try (ObjectOutputStream out = new ObjectOutputStream(
27         new FileOutputStream(args[1])))
28     {
29         out.writeObject(key);
30     }
31 }
32 else
33 {
34     int mode;
35     if (args[0].equals("-encrypt")) mode = Cipher.ENCRYPT_MODE;
36     else mode = Cipher.DECRYPT_MODE;
37
38     try (ObjectInputStream keyIn = new ObjectInputStream(
39         new FileInputStream(args[3]));
40         InputStream in = new FileInputStream(args[1]);
41         OutputStream out = new FileOutputStream(args[2]))
42     {
43         Key key = (Key) keyIn.readObject();
44         Cipher cipher = Cipher.getInstance("AES");
45         cipher.init(mode, key);
46         Util.crypt(in, out, cipher);
47     }
48 }
49 }
50 }
```

Листинг 9.19. Исходный код из файла aes/Util.java

```

1 package aes;
2
3 import java.io.*;
4 import java.security.*;
5 import javax.crypto.*;
6
7 public class Util
8 {
9     /**
10      * Использует шифр для преобразования байтов из потока ввода
11      * и направляет преобразованные байты в поток вывода
12      * @param in Поток ввода
13      * @param out Поток вывода
14      * @param cipher Шифр для преобразования байтов
15     */
16     public static void crypt(InputStream in, OutputStream out,
17         Cipher cipher) throws IOException, GeneralSecurityException
18     {
19         int blockSize = cipher.getBlockSize();
20         int outputSize = cipher.getOutputSize(blockSize);
21         byte[] inBytes = new byte[blockSize];
22         byte[] outBytes = new byte[outputSize];
23         int inLength = 0;
24
25         boolean more = true;
26         while (more)
27         {
28             inLength = in.read(inBytes);
```

```
29     if (inLength == blockSize)
30     {
31         int outLength = cipher.update(
32             inBytes, 0, blockSize, outBytes);
33         out.write(outBytes, 0, outLength);
34     }
35     else more = false;
36 }
37 if (inLength > 0) outBytes = cipher.doFinal(
38     inBytes, 0, inLength);
39 else outBytes = cipher.doFinal();
40 out.write(outBytes);
41 }
42 }
```

javax.crypto.Cipher 1.4

- **static Cipher getInstance(String algorithmName)**
- **static Cipher getInstance(String algorithmName, String providerName)**
Возвращают объект типа **Cipher**, реализующий указанный алгоритм шифрования. Если же такой алгоритм не поддерживается, то генерируется исключение типа **NoSuchAlgorithmException**.
- **int getBlockSize()**
Возвращает размер шифруемого блока в байтах или нулевое значение, если алгоритм шифрования не предусматривает манипулирование блоками данных.
- **int getOutputSize(int inputLength)**
Возвращает размер выходного буфера данных, который требуется, если следующие входные данные имеют количество байтов, определяемое параметром **inputLength**. В этом методе учитываются любые байты, буферизированные в объекте шифрования.
- **void init(int mode, Key key)**
Инициализирует объект, реализующий алгоритм шифрования. Параметр режима может принимать значение одной из следующих констант: **ENCRYPT_MODE**, **DECRYPT_MODE**, **WRAP_MODE** или **UNWRAP_MODE**.
- **byte[] update(byte[] in)**
- **byte[] update(byte[] in, int offset, int length)**
- **int update(byte[] in, int offset, int length, byte[] out)**
Преобразуют один блок входных данных. Первые два метода возвращают выходные данные, а третий метод — сведения о количестве байтов, которые были размещены в массиве **out**.
- **byte[] doFinal()**
- **byte[] doFinal(byte[] in)**
- **byte[] doFinal(byte[] in, int offset, int length)**
- **int doFinal(byte[] in, int offset, int length, byte[] out)**
Преобразуют последний блок входных данных и очищают буфер данного объекта шифрования. Первые три метода возвращают выходные данные, а четвертый метод — сведения о количестве байтов, которые были размещены в массиве **out**.

javax.crypto.KeyGenerator 1.4

- **static KeyGenerator getInstance(String algorithmName)**

Возвращает объект типа **KeyGenerator**, реализующий указанный алгоритм шифрования. Если такой алгоритм не поддерживается, то генерируется исключение типа **NoSuchAlgorithmException**.

- **void init(SecureRandom random)**

- **void init(int keySize, SecureRandom random)**

Инициализируют генератор ключей шифрования.

- **SecretKey generateKey()**

Генерирует новый ключ шифрования.

javax.crypto.spec.SecretKeySpec 1.4

- **SecretKeySpec(byte[] key, String algorithmName)**

Формирует новый секретный ключ по указанной спецификации.

9.5.3. Потоки шифрования

В библиотеке JCE имеется удобный набор классов, реализующих потоки ввода-вывода и способных автоматически шифровать и расшифровывать данные из этих потоков. В качестве примера ниже показано, как с помощью одного из таких классов зашифровать данные и вывести их в файл.

```
Cipher cipher = . . .;
cipher.init(Cipher.ENCRYPT_MODE, key);
CipherOutputStream out = new CipherOutputStream(
    new FileOutputStream(outputFileName), cipher);
byte[] bytes = new byte[BLOCKSIZE];
int inLength = getData(bytes); // получить данные из источника
while (inLength != -1)
{
    out.write(bytes, 0, inLength);
    inLength = getData(bytes);
    // получить дополнительные данные из источника
}
out.flush();
```

Подобным образом для чтения данных из файла и их расшифровки можно применить класс **CipherInputStream**:

```
Cipher cipher = . . .;
cipher.init(Cipher.DECRYPT_MODE, key);
CipherInputStream in = new CipherInputStream(
    new FileInputStream(inputFileName), cipher);
byte[] bytes = new byte[BLOCKSIZE];
int inLength = in.read(bytes);
while (inLength != -1)
{
    putData(bytes, inLength); // вывести данные по месту назначения
    inLength = in.read(bytes);
}
```

Классы потоков шифрования прозрачно обрабатывают вызовы методов `update()` и `doFinal()`, что, безусловно, очень удобно.

javax.crypto.CipherInputStream 1.4

- `CipherInputStream(InputStream in, Cipher cipher)`

Создает поток ввода, который считывает данные из потока ввода `in` и расшифровывает или зашифровывает их, используя указанный шифр.

- `int read()`

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

Считывают данные из потока ввода и автоматически расшифровывают или зашифровывают их.

javax.crypto.CipherOutputStream 1.4

- `CipherOutputStream(OutputStream out, Cipher cipher)`

Создает поток вывода, направляющий данные в заданный поток вывода `out` и зашифровывает или расшифровывает их, используя указанный шифр.

- `void write(int ch)`

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

Направляют данные в поток вывода и автоматически зашифровывают или расшифровывают их.

- `void flush()`

Выводит данные из буфера шифрования, очищая его и заполняя при необходимости недостающие биты.

9.5.4. Шифрование открытым ключом

Алгоритмы шифрования DES и AES, рассматривавшиеся в предыдущем разделе, являются *симметричными*. Это означает, что один и тот же ключ применяется как для шифрования, так и для расшифровывания. Уязвимым местом всех симметричных алгоритмов шифрования является передача ключа. Так, если Алиса отправляет Бобу зашифрованное сообщение, для его расшифровки Бобу требуется тот же самый ключ, которым пользовалась Алиса. При изменении ключа Алиса снова должна отправить Бобу вместе с сообщением новую версию этого ключа по какому-нибудь защищенному каналу. Но если у нее нет доступа к такому каналу связи с Бобом, то ей придется сначала зашифровать все отправляемые ему сообщения.

В качестве выхода из этого положения можно воспользоваться шифрованием открытым ключом. В таком случае у Боба будет пара ключей: открытый и соответствующий ему секретный. Он сможет передавать свой открытый ключ кому угодно, а секретный хранить в тайне. Алисе же останется только использовать этот открытый ключ для шифрования всех своих сообщений Бобу.

К сожалению, все не так просто. Дело в том, что все алгоритмы шифрования открытым ключом действуют *намного медленнее*, чем алгоритмы шифрования симметричными ключами вроде DES или AES. Поэтому было бы небезопасно и непрактично использовать открытые ключи для шифрования данных большого

объема. Но это затруднение нетрудно разрешить, сочетая шифрование открытым ключом с симметричным шифрованием. Обратимся за разъяснением к следующему примеру.

1. Алиса формирует произвольный симметричный ключ и шифрует им свое сообщение.
2. Используя открытый ключ Боба, Алиса зашифровывает этот симметричный ключ.
3. Алиса посыпает Бобу зашифрованное сообщение вместе с зашифрованным симметричным ключом.
4. Используя секретный ключ, Боб расшифровывает симметричный ключ.
5. Используя расшифрованный симметричный ключ, Боб расшифровывает полученное сообщение.

В данном примере никто, кроме Боба, не сможет расшифровать симметричный ключ, потому что секретный ключ имеется только у него. Таким образом, неэффективный алгоритм шифрования открытым ключом применяется только для небольшого объема данных симметричного ключа.

Наиболее распространенным для шифрования открытым ключом является алгоритм RSA, изобретенный Ривестом, Шамиром и Адлеманом. До октября 2000 года этот алгоритм был защищен патентом, выданным компанией RSA Security Inc., а для получения лицензии на его использование нужно было платить 3%-ную пошлину с минимальным ежегодным взносом 50 тыс. долларов. В настоящее время этот алгоритм уже не является коммерческим и стал всеобщим достоянием.

Чтобы воспользоваться алгоритмом шифрования RSA, необходимо создать открытый и секретный ключи средствами класса KeyPairGenerator:

```
KeyPairGenerator pairgen = KeyPairGenerator.getInstance("RSA");
SecureRandom random = new SecureRandom();
pairgen.initialize(KEYSIZE, random);
KeyPair keyPair = pairgen.generateKeyPair();
Key publicKey = keyPair.getPublic();
Key privateKey = keyPair.getPrivate();
```

Для запуска на выполнение примера программы из листинга 9.20 применяются три параметра. В частности, параметр **-genkey** служит для создания пары ключей, а параметр **-encrypt** — для генерирования ключа по алгоритму шифрования AES и его свертывания с помощью открытого ключа, как показано ниже.

```
Key key = . . .; // ключ по алгоритму шифрования AES
Key publicKey = . . .; // открытый ключ по алгоритму шифрования RSA
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.WRAP_MODE, publicKey);
byte[] wrappedKey = cipher.wrap(key);
```

Далее создается файл, состоящий из следующих элементов.

- Длина свернутого ключа.
- Байты свернутого ключа.
- Открытый текст, зашифрованный ключом по алгоритму AES.

Третий параметр, **-decrypt**, служит для расшифровывания файла. Чтобы опробовать данную программу, необходимо сначала создать ключи по алгоритму шифрования RSA, выполнив следующую команду:

```
java rsa.RSATest -genkey public.key private.key
```

Затем следует зашифровать файл по команде:

```
java rsa.RSATest -encrypt plaintextFile encryptedFile public.key
```

И наконец, файл следует расшифровать и проверить, совпадают ли полученные данные с ранее зашифрованными, введя приведенную ниже команду.

```
java rsa.RSATest -decrypt encryptedFile decryptedFile private.key
```

Листинг 9.20. Исходный код из файла rsa/RSATest.java

```
1 package rsa;
2
3 import java.io.*;
4 import java.security.*;
5 import javax.crypto.*;
6 /**
7  * В этой программе проверяется шифрование по алгоритму RSA.
8  * Применение:
9  *   java rsa.RSATest -genkey public private
10 *   java rsa.RSATest -encrypt plaintext encrypted public
11 *   java rsa.RSATest -decrypt encrypted decrypted private
12 * @author Cay Horstmann
13 * @version 1.01 2012-06-10
14 */
15 public class RSATest
16 {
17     private static final int KEYSIZE = 512;
18
19     public static void main(String[] args) throws IOException,
20                             GeneralSecurityException, ClassNotFoundException
21     {
22         if (args[0].equals("-genkey"))
23         {
24             KeyPairGenerator pairgen =
25                 KeyPairGenerator.getInstance("RSA");
26             SecureRandom random = new SecureRandom();
27             pairgen.initialize(KEYSIZE, random);
28             KeyPair keyPair = pairgen.generateKeyPair();
29             try (ObjectOutputStream out = new ObjectOutputStream(
30                  new FileOutputStream(args[1])))
31             {
32                 out.writeObject(keyPair.getPublic());
33             }
34             try (ObjectOutputStream out = new ObjectOutputStream(
35                  new FileOutputStream(args[2])))
36             {
37                 out.writeObject(keyPair.getPrivate());
38             }
39         }
40         else if (args[0].equals("-encrypt"))
41         {
```

```

42     KeyGenerator keygen = KeyGenerator.getInstance("AES");
43     SecureRandom random = new SecureRandom();
44     keygen.init(random);
45     SecretKey key = keygen.generateKey();
46
47     // свернуть с помощью открытого ключа по алгоритму RSA
48     try (ObjectInputStream keyIn = new ObjectInputStream(
49             new FileInputStream(args[3]));
50         DataOutputStream out = new DataOutputStream(
51             new FileOutputStream(args[2]));
52         InputStream in = new FileInputStream(args[1])) {
53
54         Key publicKey = (Key) keyIn.readObject();
55         Cipher cipher = Cipher.getInstance("RSA");
56         cipher.init(Cipher.WRAP_MODE, publicKey);
57         byte[] wrappedKey = cipher.wrap(key);
58         out.writeInt(wrappedKey.length);
59         out.write(wrappedKey);
60
61         cipher = Cipher.getInstance("AES");
62         cipher.init(Cipher.ENCRYPT_MODE, key);
63         Util.crypt(in, out, cipher);
64     }
65 } else {
66
67     try (DataInputStream in = new DataInputStream(
68             new FileInputStream(args[1]));
69             ObjectInputStream keyIn =
70             new ObjectInputStream(new FileInputStream(args[3])));
71             OutputStream out = new FileOutputStream(args[2])) {
72
73         int length = in.readInt();
74         byte[] wrappedKey = new byte[length];
75         in.read(wrappedKey, 0, length);
76
77         // развернуть с помощью секретного ключа по алгоритму RSA
78         Key privateKey = (Key) keyIn.readObject();
79
80         Cipher cipher = Cipher.getInstance("RSA");
81         cipher.init(Cipher.UNWRAP_MODE, privateKey);
82         Key key = cipher.unwrap(wrappedKey, "AES",
83             Cipher.SECRET_KEY);
84
85         cipher = Cipher.getInstance("AES");
86         cipher.init(Cipher.DECRYPT_MODE, key);
87
88         Util.crypt(in, out, cipher);
89     }
90 }
91 }
92 }
93 }
```

В этой главе было показано, каким образом модель безопасности обеспечивает контролируемое выполнение кода, что является отличительной и очень важной особенностью платформы Java. Кроме того, в этой главе были рассмотрены различные службы, предоставляемые в библиотеке Java для аутентификации

и шифрования. Тем не менее имеется еще немало дополнительных и специальных вопросов безопасности, которые не были затронуты в этой главе.

- Прикладной интерфейс GSS API для “универсальных служб безопасности”, обеспечивающий поддержку протокола Kerberos (и, в принципе, других протоколов, предназначенных для безопасного обмена сообщениями). Подробнее об этом прикладном интерфейсе можно узнать по адресу <http://docs.oracle.com/javase/8/docs/technotes/guides/security/jgss/tutorials>.
- Поддержка механизма SASL (Simple Authentication and Security Layer — простой уровень аутентификации и безопасности), применяемого в сетевых протоколах LDAP и IMAP. Если вам потребуется реализовать механизм SASL в своей прикладной программе, необходимые сведения по данному вопросу вы найдете по адресу <http://docs.oracle.com/javase/8/docs/technotes/guides/security/sasl-refguide.html>.
- Поддержка сетевого протокола SSL (Secure Sockets Layer — уровень защищенных сокетов). Применение протокола SSL через сетевой протокол HTTP является вполне прозрачным для разработчиков прикладных программ и состоит в том, чтобы указывать префикс `https` в URL. Если вам потребуется ввести протокол SSL в свою прикладную программу, необходимые сведения по данному вопросу вы можете найти в документе JSSE (Java Secure Socket Extension — Расширение Java для защищенных сокетов) по адресу <http://docs.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html>.

В следующей главе будут рассмотрены расширенные возможности для программирования средствами библиотеки Swing.

Расширенные средства Swing

В этой главе...

- ▶ Списки
- ▶ Таблицы
- ▶ Деревья
- ▶ Текстовые компоненты
- ▶ Индикаторы состояния
- ▶ Организаторы и декораторы компонентов

В этой главе продолжается начатое в первом томе обсуждение набора компонентов из библиотеки Swing, предназначенных для создания графического пользовательского интерфейса (ППИ). В первом томе настоящего издания были представлены только самые основные и наиболее употребительные компоненты, хотя в библиотеке Swing имеется немало других намного более сложных компонентов. Сначала в этой главе рассматриваются списки, таблицы и деревья, которым посвящена большая ее часть. Затем в ней описываются текстовые компоненты и приводятся дополнительные сведения о простых текстовых полях и областях, о которых уже упоминалось в первом томе. По ходу изложения материала главы будет показано, каким образом в текстовые поля добавляются средства проверки достоверности вводимых данных и счетчики, а также отображается структурированный текст, например в формате HTML. Далее будут рассмотрены компоненты для индикации хода выполнения длительных операций. И в завершение главы будут представлены организаторы компонентов в виде панелей с вкладками и настольных панелей с внутренними фреймами.

10.1. Списки

Чтобы предоставить пользователю возможность выбирать среди нескольких значений параметров, вместо групп кнопок-переключателей или флажков (которые занимают довольно много места) можно воспользоваться обычными или комбинированными списками. Комбинированные списки имеют довольно простую структуру и подробно рассматриваются в первом томе настоящего издания. А компонент `JList`, реализующий списки, обладает намного более развитыми средствами для создания сложных структур, подобных деревьям и таблицам. Именно поэтому в этой главе сначала рассматриваются простые списки.

Чаще всего применяются списки, состоящие из символьных строк, но элементами списка могут быть произвольные объекты, допускающие управление своим внешним видом. Компонент `JList` обладает изящной внутренней структурой, благодаря которой и достигается подобная универсальность. К сожалению, разработчики библиотеки `Swing` уделили чрезмерно большое внимание изящности структуры данного компонента в ущерб простоте его использования. В ряде простых ситуаций компонент `JList` оказывается не очень удобным в употреблении, поскольку он вынуждает пользоваться универсальными механизмами. Сначала в этом разделе рассматривается простой и очень распространенный пример организации списка символьных строк, а затем более сложный пример создания списка, который наглядно демонстрирует гибкость компонента `JList`.

10.1.1. Компонент `JList`

Этот компонент служит для отображения ряда элементов в одном окне со списком. На рис. 10.1 показан очень простой пример списка, в котором пользователь может подобрать характеристики для описания лисы: `quick` (шустрая), `brown` (бурая), `hungry` (голодная), `wild` (дикая), а также `static`, `private` и `final` (тут мы исчерпали известные нам характеристики животных и предложили первые пришедшие на память ключевые слова Java). Эти характеристики будут использованы в предложении о лисе, которая перепрыгивает через ленивую собаку. Например, при выборе в списке характеристик `static` и `final` получится предложение `The static final fox jumps over the lazy dog`, которое появится в рабочем окне рассматриваемого здесь примера программы.

В версии Java SE 7 компонент `JList` стал обобщенным. В качестве параметра типа в нем служат типы значений, которые может выбирать пользователь. В рассматриваемом здесь примере используется строковый тип `JList<String>`.

Составление простого списка следует начинать с создания массива символьных строк, а затем передать его конструктору компонента `JList` следующим образом:

```
String[] words = { "quick", "brown", "hungry", "wild", . . . };
JList<String> wordList = new JList<>(words);
```

В простых списках не предусмотрены средства автоматической прокрутки. Поэтому для просмотра длинного списка его нужно расположить на панели с полосой прокрутки типа `JScrollPane`, как показано ниже, а затем вставить эту панель (а не список) в другую, окружающую ее панель.

```
JScrollPane scrollPane = new JScrollPane(wordList);
```

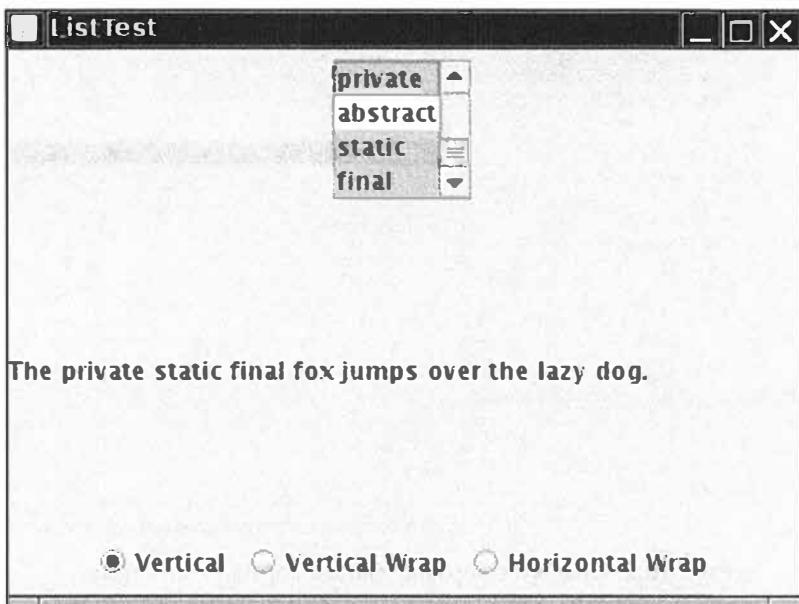


Рис. 10.1. Пример простого списка

Следует, однако, признать, что разделение функций отображения списка и механизма прокрутки выглядит изящно теоретически, но неудобно на практике, так как редкий список обходится без прокрутки его содержимого. Поэтому совершенно неоправданно заставлять разработчиков программировать самостоятельно те функции, которые могли бы быть реализованы по умолчанию, только для того, чтобы они оценили универсальность и изящество структуры компонента `JList`.

По умолчанию в окне списка отображаются восемь элементов. Но это количество можно изменить с помощью метода `setVisibleRowCount()`, как показано в приведенном ниже примере.

```
wordList.setVisibleRowCount(4); // отобразить четыре элемента списка
```

При составлении списка можно выбрать один из трех вариантов расположения его элементов.

- **`JList.VERTICAL` (по умолчанию).** Все элементы списка располагаются вертикально один под другим.
- **`JList.VERTICAL_WRAP`.** Если количество элементов списка превышает заданное число отображаемых символьных строк, формируются новые столбцы (рис. 10.2).
- **`JList.HORIZONTAL_WRAP`.** Если количество элементов превышает заданное число отображаемых символьных строк, формируются новые столбцы, но они заполняются в горизонтальном направлении. Отличие между вариантами расположения `JList.VERTICAL_WRAP` и `JList.HORIZONTAL_WRAP` нетрудно понять, сравнив на рис. 10.2 расположение элементов `quick`, `brown` и `hungry`.

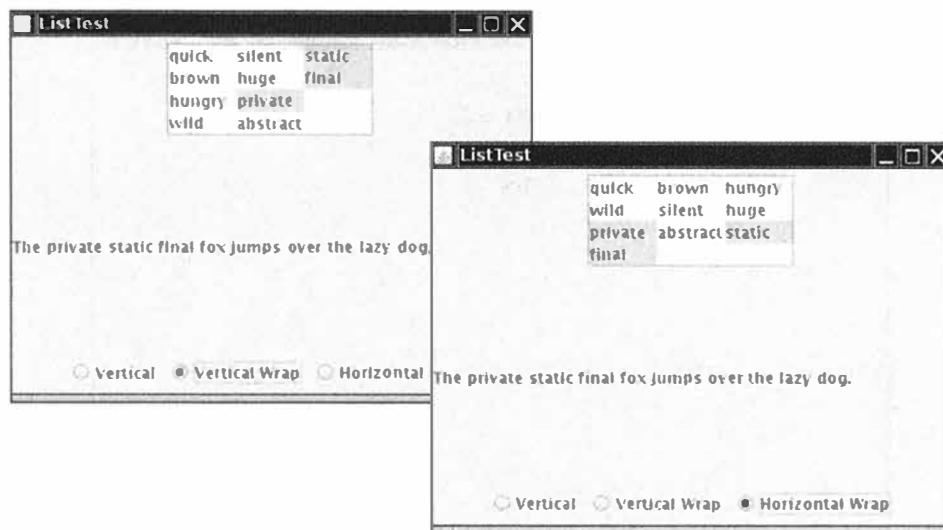


Рис. 10.2. Списки с вертикальным горизонтальным расположением элементов в дополнительных столбцах

По умолчанию пользователь может выбрать сразу несколько элементов из списка. Например, для выбора несмежных элементов из списка следует нажать клавишу **<Ctrl>** и, не отпуская ее, щелкнуть на них по очереди кнопкой мыши. А для выбора нескольких смежных элементов из списка следует нажать клавишу **<Shift>** и, не отпуская ее, щелкнуть кнопкой мыши сначала на первом из них, а затем на последнем.

Для ограничения вариантов выбора из списка предусмотрен метод `setSelectionMode()`. В приведенном ниже фрагменте кода показано, каким образом накладывается подобное ограничение.

```
wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    // разрешить выбор элементов по одному
wordList.setSelectionMode(
    ListSelectionModel.SINGLE_INTERVAL_SELECTION);
    // разрешить выбор одного элемента или ряда элементов
```

Как упоминалось в первом томе настоящего издания, базовые компоненты пользовательского интерфейса посыпают сообщения о событиях, связанных с действиями пользователя. А в списках используется другой механизм уведомления о наступающих событиях. Вместо события действия при обращении к спискам следует принимать событие выбора из списка. Для этого в компонент составления списков нужно ввести приемник событий выбора из списка и реализовать в нем следующий метод:

```
public void valueChanged(ListSelectionEvent evt)
```

Когда пользователь выбирает элемент из списка, инициируется несколько событий. Допустим, пользователь щелкнул на элементе списка. При нажатии кнопки мыши передается событие об изменении выбора. Это промежуточное

событие. Приведенный ниже метод возвращает логическое значение `true`, если выбор не был окончательным.

```
event.getValueIsAdjusting()
```

При отпускании кнопки мыши генерируется еще одно событие. На этот раз метод `isAdjusting()` возвращает логическое значение `false`. Если обработка промежуточных событий не нужна, то отслеживать можно только те события, для которых метод `isAdjusting()` возвращает логическое значение `false`. Но для организации быстрого реагирования на действия пользователя с мышью придется обрабатывать все события.

После уведомления о событии необходимо выяснить, какие именно элементы были выбраны из списка. Если список находится в режиме выбора только одного элемента, следует вызвать метод `getSelectedValue()`, чтобы получить значение, обозначающее тип элемента, выбранного из списка. В противном случае следует вызвать метод `getSelectedValuesList()`, возвращающий список, содержащий все выбранные элементы. Полученный результат обрабатывается обычным образом, как показано ниже.

```
for (String value : wordList.getSelectedValuesList())
    // сделать что-нибудь с полученным значением value
```



На заметку! В компоненте `JList` не предусмотрено реагирование на двойной щелчок кнопкой мыши. Разработчики библиотеки Swing предполагали, что список предназначается только для выбора элементов, а для выполнения какого-нибудь другого действия следует щелкнуть на соответствующей кнопке. Но в некоторых реализациях пользовательского интерфейса двойной щелчок кнопкой мыши рассматривается одновременно как выбор элемента и команда на выполнение соответствующего действия. Поэтому если требуется предоставить пользователю такую возможность, необходимо сначала связать со списком обработчик событий от мыши, а затем перехватывать и обрабатывать подобные события, как показано ниже.

```
public void mouseClicked(MouseEvent evt)
{
    if (evt.getClickCount() == 2)
    {
        JList source = (JList) evt.getSource();
        Object[] selection = source.getSelectedValues();
        doAction(selection);
    }
}
```

В листинге 10.1 приведен исходный код примера программы, где демонстрируется применение списка, элементами которого являются символьные строки. Обратите внимание на метод `valueChanged()`, формирующий символьную строку сообщения из элементов, выбираемых из списка.

Листинг 10.1. Исходный код из файла `list/ListFrame.java`

```
1 package list;
2
3 import java.awt.*;
```

```
5 import javax.swing.*;
6
7 /**
8  * Этот фрейм содержит список слов и метку, отображающую
9  * предложение, составляемое из слов, выбираемых из данного
10 * списка. Чтобы выбрать сразу несколько слов из списка, можно
11 * воспользоваться комбинациями <Ctrl+щелчок> и <Shift+щелчок>
12 */
13
14 class ListFrame extends JFrame
15 {
16     private static final int DEFAULT_WIDTH = 400;
17     private static final int DEFAULT_HEIGHT = 300;
18
19     private JPanel listPanel;
20     private JList<String> wordList;
21     private JLabel label;
22     private JPanel buttonPanel;
23     private ButtonGroup group;
24     private String prefix = "The ";
25     private String suffix = "fox jumps over the lazy dog.";
26
27     public ListFrame()
28     {
29         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
30
31         String[] words = { "quick", "brown", "hungry", "wild",
32                            "silent", "huge", "private",
33                            "abstract", "static", "final" };
34
35         wordList = new JList<>(words);
36         wordList.setVisibleRowCount(4);
37         JScrollPane scrollPane = new JScrollPane(wordList);
38
39         listPanel = new JPanel();
40         listPanel.add(scrollPane);
41         wordList.addListSelectionListener(event ->
42         {
43             StringBuilder text = new StringBuilder(prefix);
44             for (String value : wordList.getSelectedValuesList())
45             {
46                 text.append(value);
47                 text.append(" ");
48             }
49             text.append(suffix);
50
51             label.setText(text.toString());
52         });
53
54         buttonPanel = new JPanel();
55         group = new ButtonGroup();
56         makeButton("Vertical", JList.VERTICAL);
57         makeButton("Vertical Wrap", JList.VERTICAL_WRAP);
58         makeButton("Horizontal Wrap", JList.HORIZONTAL_WRAP);
59
60         add(listPanel, BorderLayout.NORTH);
61         label = new JLabel(prefix + suffix);
62         add(label, BorderLayout.CENTER);
63         add(buttonPanel, BorderLayout.SOUTH);
```

```
64    }
65
66    /**
67     * Создает кнопку-переключатель для выбора варианта
68     * расположения элементов в списке
69     * @param label Метка кнопки
70     * @param orientation Вариант расположения элементов в списке
71     */
72    private void makeButton(String label, final int orientation)
73    {
74        JRadioButton button = new JRadioButton(label);
75        buttonPanel.add(button);
76        if (group.getButtonCount() == 0) button.setSelected(true);
77        group.add(button);
78        button.addActionListener(event ->
79        {
80            wordList.setLayoutOrientation(orientation);
81            listPanel.revalidate();
82        });
83    }
84 }
```

`javax.swing.JList<E>` 1.2

- **`JList(E[] items)`**

Составляет список из указанных элементов для их отображения.

- **`int getVisibleRowCount()`**

- **`void setVisibleRowCount(int c)`**

Устанавливают или получают количество строк с элементами, которые можно увидеть в списке без прокрутки.

- **`int getLayoutOrientation() 1.4`**

- **`void setLayoutOrientation(int orientation) 1.4`**

Устанавливают или получают вариант расположения элементов в списке.

Параметры:

`orientation`

Принимает значение одной

из следующих констант:

`VERTICAL`, **`VERTICAL_WRAP`**

или **`HORIZONTAL_WRAP`**

- **`int getSelectionMode()`**

- **`void setSelectionMode(int mode)`**

Устанавливают или получают режим, определяющий возможность выбрать сразу один или несколько элементов из списка.

Параметры:

`mode`

Принимает значение одной из

следующих констант:

`SINGLE_SELECTION`,

`SINGLE_INTERVAL_SELECTION`,

`MULTIPLE_INTERVAL_SELECTION`

javax.swing.JList<E> 1.2 (окончание)

- **void addListSelectionListener(ListSelectionListener listener)**
Связывает со списком приемник событий, который уведомляется о каждом изменении выбора элементов из списка.
- **List<E> getSelectedValues()** 7
Возвращает выбранные из списка значения или пустой список, если ничего не выбрано.
- **E getSelectedValue()**
Возвращает первое выбранное из списка значение или пустое значение **null**, если ничего не выбрано.

javax.swing.event.ListSelectionListener 1.2

- **void valueChanged(ListSelectionEvent e)**
Вызывается при каждом изменении выбора элементов из списка.

10.1.2. Модели списков

В предыдущем разделе описан наиболее употребительный способ применения компонента **JList** для составления списков. В частности, для составления простого списка необходимо выполнить следующие действия.

1. Указать фиксированный ряд символьных строк для отображения в списке.
2. Расположить список на панели, снабженной полосой прокрутки.
3. Организовать обработку событий, наступающих при выборе элементов из списка.

А далее рассматриваются следующие более сложные варианты списков.

- Очень длинные списки.
- Списки с изменяющимся содержимым.
- Списки, состоящие из других элементов, кроме символьных строк.

В предыдущем примере рассматривалось создание с помощью компонента **JList** списка с фиксированным набором символьных строк. Но набор элементов списка может изменяться. Как же организовать ввод и удаление элементов из списка? Как ни странно, в классе **JList** отсутствуют методы для подобных целей. Поэтому придется более подробно рассмотреть внутреннюю структуру списка. В частности, для создания списков в компоненте **JList** применяется проектный шаблон “модель–представление–контроллер” (MVC), чтобы отделить базовые данные (т.е. набор объектов) от их визуального представления (т.е. отображения элементов списка столбцами).

Класс **JList** отвечает за визуальное представление данных, и ему ничего неизвестно о способе их хранения. Ему известно только, что данные можно извлечь, обратившись к объекту из класса, реализующего приведенный ниже интерфейс **ListModel**:

```
public interface ListModel<E>
{
    int getSize();
    E getElementAt(int i);
    void addListDataListener(ListDataListener l);
    void removeListDataListener(ListDataListener l);
}
```

Компонент `JList` может получать через этот интерфейс подсчет элементов списка и извлекать любой из них. Кроме того, объект типа `JList` может выступать в роли приемника событий типа `ListDataListener`. В этом случае он уведомляется об изменении состава списка, чтобы перерисовать его. В чем же польза от такой универсальности? Почему бы не поступить проще, сохраняя в объекте типа `JList` массив других объектов?

Дело в том, что в интерфейсе `ListModel` не указывается способ хранения данных и даже не требуется вообще сохранять их! В частности, метод `getElementAt()` можно переопределить таким образом, чтобы он вычислял требуемое значение всякий раз, когда он вызывается. Такой способ удобен для представления очень большого набора данных без сохранения значений.

Рассмотрим еще один, несколько упрощенный пример составления списка, из которого пользователь может выбрать любой из всех возможных вариантов трехбуквенных слов на английском языке (рис. 10.3). В английском языке насчитывается $26 \times 26 \times 26 = 17576$ трехбуквенных сочетаний. Вместо хранения всех этих сочетаний было бы разумнее вычислять их во время прокрутки списка.

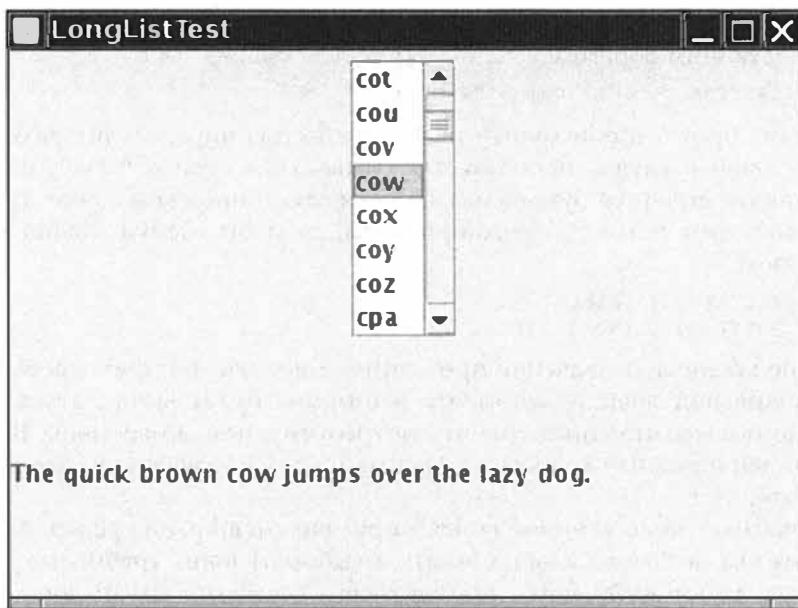


Рис. 10.3. Пример выбора варианта из очень длинного списка

Оказывается, что составить такой список совсем не трудно. Рутинные операции ввода и удаления приемников событий уже реализованы в классе

`AbstractListModel`, который можно расширить конкретным классом. Остается только предоставить методы `getSize()` и `getElementAt()`, как показано ниже. Подробнее о вычислении *n*-й символьной строки см. в исходном коде из листинга 10.3.

```
class WordListModel extends AbstractListModel<String>
{
    public WordListModel(int n) { length = n; }
    public int getSize() { return (int) Math.pow(26, length); }
    public String getElementAt(int n)
    {
        // вычислить n-ю символьную строку
        . . .
    }
    . . .
}
```

Имея теперь модель, можно составить список, который можно прокручивать, просматривая и выбирая элементы, предоставляемые данной моделью, как показано в приведенном ниже фрагменте кода. Основной замысел состоит в том, что символьные строки не сохраняются, а вычисляются, причем формируются только те символьные строки, которые требуются пользователю в данный момент.

```
JList<String> wordList = new JList<>(new WordListModel(3));
wordList.setSelectionMode(ListSelectionMode.SINGLE_SELECTION);
JScrollPane scrollPane = new JScrollPane(wordList);
```

Необходимо сделать еще одну установку: уведомить компонент составления списков о том, что все элементы списка имеют фиксированную высоту и ширину. Установить размеры ячейки проще всего, определив значение *прототипной ячейки* следующим образом:

```
wordList.setPrototypeCellValue("www");
```

Значение прототипной ячейки используется для определения размеров всех ячеек. (В данном случае используется символьная строка "www", потому что в большинстве шрифтов буква *w* оказывается самой широкой из всех прописных букв английского языка.) С другой стороны, размеры ячейки можно задать явным образом:

```
wordList.setFixedCellWidth(50);
wordList.setFixedCellHeight(15);
```

Если не установить значение прототипной ячейки или фиксированные размеры конкретной ячейки, ее высота и ширина будут вычисляться отдельно для каждого элемента списка, на что потребуется немало времени. В листинге 10.2 приведен исходный код класса фрейма из рассматриваемого здесь примера программы.

На практике очень длинные списки применяются крайне редко, потому что пользователям не совсем удобно искать и выбирать в них требуемые элементы. Кроме того, набор выбранных пользователем элементов имеет достаточно малый объем и может быть сохранен непосредственно в компоненте `JList`. Такой механизм избавляет от необходимости обращаться к модели списка отдельно от самого компонента. С другой стороны, класс `JList` оказывается совместимым с классами `JTable` и `JTree`.

Листинг 10.2. Исходный код из файла longList/LongListFrame.java

```
1 package longList;
2
3 import java.awt.*;
4
5 import javax.swing.*;
6
7 /**
8  * Этот фрейм содержит длинный список слов и метку, отображающую
9  * предложение, составляемое из слов, выбираемых из данного списка
10 */
11 public class LongListFrame extends JFrame
12 {
13     private JList<String> wordList;
14     private JLabel label;
15     private String prefix = "The quick brown ";
16     private String suffix = " jumps over the lazy dog.";
17
18     public LongListFrame()
19     {
20         wordList = new JList<String>(new WordListModel(3));
21         wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
22         wordList.setPrototypeCellValue("www");
23         JScrollPane scrollPane = new JScrollPane(wordList);
24
25         JPanel p = new JPanel();
26         p.add(scrollPane);
27         wordList.addListSelectionListener(event ->
28             setSubject(wordList.getSelectedValue()));
29
30         Container contentPane = getContentPane();
31         contentPane.add(p, BorderLayout.NORTH);
32         label = new JLabel(prefix + suffix);
33         contentPane.add(label, BorderLayout.CENTER);
34         setSubject("fox");
35         pack();
36     }
37
38 /**
39  * Устанавливает в метке новый субъект
40  * @param word Новый субъект, перепрыгивающий
41  *             через ленивую собаку
42  */
43     public void setSubject(String word)
44     {
45         StringBuilder text = new StringBuilder(prefix);
46         text.append(word);
47         text.append(suffix);
48         label.setText(text.toString());
49     }
50 }
```

Листинг 10.3. Исходный код из файла longList/WordListModel.java

```
1 package longList;
2
3 import javax.swing.*;
```

```

4
5  /**
6   * Модель, динамически формирующая слова из л букв
7  */
8 public class WordListModel extends AbstractListModel<String>
9 {
10    private int length;
11    public static final char FIRST = 'a';
12    public static final char LAST = 'z';
13
14   /**
15    * Конструирует модель
16    * @param n Длина слова
17   */
18   public WordListModel(int n)
19   {
20     length = n;
21   }
22
23   public int getSize()
24   {
25     return (int) Math.pow(LAST - FIRST + 1, length);
26   }
27
28   public String getElementAt(int n)
29   {
30     StringBuilder r = new StringBuilder();
31     for (int i = 0; i < length; i++)
32     {
33       char c = (char) (FIRST + n % (LAST - FIRST + 1));
34       r.insert(0, c);
35       n = n / (LAST - FIRST + 1);
36     }
37     return r.toString();
38   }
39 }

```

javax.swing.JList<E> 1.2

- **JList(ListModel<E> dataModel)**

Составляет список, в котором отображаются элементы из указанной модели.

- **E getPrototypeCellValue()**

- **void setPrototypeCellValue(E newValue)**

Получают или устанавливают значение прототипной ячейки, используемое для определения ширины и высоты каждой ячейки в списке. По умолчанию принимается пустое значение `null`, в результате чего размеры вычисляются отдельно для каждого элемента списка.

- **void setFixedCellWidth(int width)**

- **void setFixedCellHeight(int height)**

Если значение параметра `width` или `height` оказываются больше нуля, эти методы определяют ширину или высоту (в пикселях) всех ячеек в списке. По умолчанию параметр `width` или `height` принимает значение `-1`, которое означает автоматическое определение размеров каждой ячейки.

```
javax.swing.ListModel<E> 1.2
```

- **int getSize()**

Возвращает количество элементов в модели списка.

- **E getElementAt(int position)**

Возвращает из модели списка элемент, находящийся на указанной позиции.

10.1.3. Ввод и удаление значений

Непосредственно редактировать набор значений в списке нельзя. Для этого нужно сначала обеспечить доступ к модели и только затем выполнять операции удаления и добавления элементов в список. Но это проще сказать, чем сделать. Итак, сначала необходимо получить ссылку на модель списка следующим образом:

```
ListModel<String> model = list.getModel();
```

Но это ничего не дает. Ведь, как упоминалось ранее, в интерфейсе ListModel отсутствуют методы добавления и удаления элементов из списка. Основное назначение модели списка состоит в том, чтобы исключить потребность в хранении элементов. Попробуем поступить иначе. Один из конструкторов класса JList получает в качестве параметра вектор объектов:

```
Vector<String> values = new Vector<String>();
values.addElement("quick");
values.addElement("brown");
.
.
JList<String> list = new JList<>(values);
```

Теперь можно редактировать вектор, добавляя и удаляя из него элементы. Но списку ничего не известно о подобных изменениях, поэтому он никак не отреагирует на них. В частности, список не будет обновляться при добавлении новых элементов. Следовательно, данный конструктор не приносит особой пользы.

Вместо этого следует создать объект типа DefaultListModel, заполнить его нужными значениями и связать со списком, как показано в приведенном ниже фрагменте кода. Класс DefaultListModel реализует интерфейс ListModel и управляет коллекцией объектов.

```
DefaultListModel<String> model = new DefaultListModel<>();
model.addElement("quick");
model.addElement("brown");
.
.
JList<String> list = new JList<>(model);
```

Теперь в объект model можно добавлять значения и удалять их из него, как следует из приведенного ниже фрагмента кода. Этот объект будет уведомлять список о происходящих изменениях, а список, в свою очередь, изменять отображаемые данные.

```
model.removeElement("quick");
model.addElement("slow");
```

Исторически сложилось так, что методы из класса DefaultListModel отличаются своими именами от методов из классов коллекций. Для внутреннего

хранения значений в исходной модели списка типа `DefaultListModel` используется вектор.

Внимание! Некоторые конструкторы класса `JList` создают список из массива или вектора объектов или символьных строк. На первый взгляд, эти конструкторы используют класс `DefaultListModel` для хранения значений элементов. Но это совсем не так. Они создают обычную модель, в которой отсутствует уведомление об изменении содержимого. Ниже приводится исходный код конструктора для создания списка типа `JList` из объекта типа `Vector`.

```
public JList(final Vector<? extends E> listData)
{
    this (new AbstractListModel<E>()
    {
        public int getSize() { return listData.size(); }
        public E getElementAt(int i) { return listData.elementAt(i); }
    });
}
```

Это означает, что если вектор изменится после создания списка, то в списке могут вспрятаться старые и новые значения до тех пор, пока он не будет полностью воспроизведен снова. (Ключевое слово `final` в приведенном выше конструкторе не накладывает запрет на изменение самого вектора, а только запрещает изменять значение ссылки на объект типа `listData`. Это ключевое слово требуется, потому что объект типа `listData` используется во внутреннем классе.)

javax.swing.JList<E> 1.2

- `ListModel<E> getModel()`
Возвращает модель для данного списка.

javax.swing.DefaultListModel<E> 1.2

- `void addElement(E obj)`
Добавляет объект в конце модели.
- `boolean removeElement(Object obj)`
Удаляет из модели первый экземпляр объекта и возвращает логическое значение `true`, если такой объект обнаружен, а иначе — логическое значение `false`.

10.1.4. Воспроизведение значений

Все рассмотренные до сих пор списки содержали символьные строки. Но список можно также составить из пиктограмм, передав ему массив или вектор объектов типа `Icon`. Более того, список можно составить из любых объектов и воспроизвести их любым требующимся способом.

И хотя класс `JList` позволяет воспроизвести символьные строки и пиктограммы в списке автоматически, для воспроизведения произвольных объектов необходимо установить в объекте типа `JList` средство воспроизведения ячеек из списка.

В качестве такого средства может выступать экземпляр любого класса, реализующего приведенный ниже интерфейс `ListCellRenderer`.

```
interface ListCellRenderer<E>
{
    Component getListCellRendererComponent(JList<? extends E> list,
                                           E value, int index, boolean isSelected, boolean cellHasFocus);
}
```

Метод, объявленный в данном интерфейсе, вызывается для каждой ячейки. Он возвращает компонент, предназначенный для воспроизведения содержимого ячейки. Этот компонент размещается там, где предполагается воспроизвести ячейку. Чтобы реализовать средство воспроизведения ячеек из списка, проще всего создать класс, расширяющий класс `JComponent`. Пример такого класса приведен ниже.

```
class MyCellRenderer extends JComponent implements ListCellRenderer<Тип>
{
    public Component getListCellRendererComponent(
        JList<? extends Тип> list, Тип value,
        int index, boolean isSelected,
        boolean cellHasFocus)
    {
        // сохранить данные, необходимые для воспроизведения
        // и определения размеров ячеек
        return this;
    }
    public void paintComponent(Graphics g)
    {
        // здесь следует код для воспроизведения ячеек
    }
    public Dimension getPreferredSize()
    {
        // здесь следует код для определения размеров ячеек
    }
    // поля экземпляра
}
```

В листинге 10.4 приведен пример программы для графического отображения шрифтов, выбираемых из списка (рис. 10.4). Для вывода названия каждого шрифта вызывается метод `paintComponent()`. При этом необходимо учитывать цвета воспроизводимого списка. Требуемые сведения о цвете переднего и заднего планов получаются с помощью методов `getForeground()/getBackground()` и `getSelectionForeground()/getSelection Background()` из класса `JList`. В методе `getPreferredSize()` определяются типографские размеры воспроизводимой символьной строки способом, подробно описанным в главе 10 первого тома настоящего издания.

Для установки средства воспроизведения ячеек из списка достаточно вызвать приведенный ниже метод `setCellRenderer()`. В итоге все ячейки будут выводиться из списка с помощью специально установленного средства их воспроизведения.

```
fontList.setCellRenrderer(new FontCellRenderer);
```

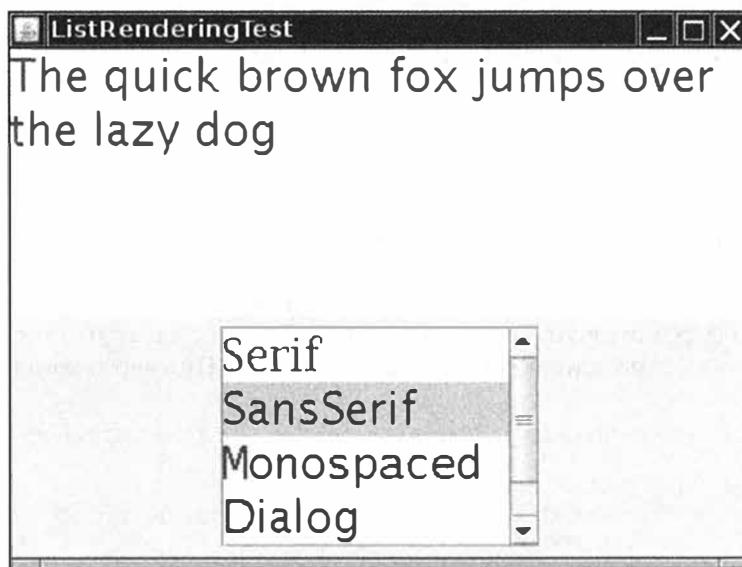


Рис. 10.4. Пример списка с воспроизведением ячеек разными шрифтами

Существует более простой способ создания специальных средств воспроизведения ячеек из списка. Так, если изображение содержит текст, пиктограмму, а возможно, и меняющийся цвет, то для его воспроизведения можно обойтись соответственно настроенными средствами класса `JLabel`. Например, для выделения названия шрифта тем же самым шрифтом подходит следующее средство воспроизведения:

```
class FontCellRenderer extends JLabel implements
    ListCellRenderer<Font>
{
    public Component getListCellRendererComponent(
        JList<? extends Font> list,
        Font value, int index, boolean isSelected,
        boolean cellHasFocus)
    {
        Font font = (Font) value;
        setText(font.getFamily());
        setFont(font);
        setOpaque(true);
        setBackground(isSelected ? list.getSelectionBackground() :
            list.getBackground());
        setForeground(isSelected ? list.getSelectionForeground() :
            list.getForeground());
        return this;
    }
}
```

В данном случае не пришлось самостоятельно создавать методы `paintComponent()` или `getPreferredSize()`, поскольку они уже реализованы в классе `JLabel`. Поэтому остается лишь задать параметры метки, т.е. указать текст, шрифт и цвет. Такой код оказывается удобным в тех случаях, когда

какой-нибудь существующий компонент (в данном случае `JLabel`) поддерживает все функциональные возможности, требующиеся для воспроизведения содержимого ячеек из списка.

В рассматриваемом здесь примере программы можно было бы воспользоваться компонентом `JLabel`, но ее исходный код специально написан обобщенным, чтобы в него можно было легко внести изменения, если в ячейках из составляемого списка потребуется воспроизвести что-нибудь другое.

! **На заметку!** Создавать новый компонент при каждом вызове метода `getListCellRendererComponent()` вряд ли оправдано. Так, если пользователю приходится прокручивать список, состоящий из большого количества элементов, то всякий раз будет создаваться новый компонент. Более эффективное и надежное решение состоит в перенастройке существующего компонента.

Листинг 10.4. Исходный код из файла listRendering/FontCellRenderer.java

```
1 package listRendering;
2
3 import java.awt.*;
4 import javax.swing.*;
5 /**
6  * Средство воспроизведения названий шрифтов самими шрифтами,
7  * выбираемыми в виде объектов типа Font из ячеек списка
8 */
9 public class FontCellRenderer extends JComponent
10    implements ListCellRenderer<Font>
11 {
12     private Font font;
13     private Color background;
14     private Color foreground;
15
16     public Component getListCellRendererComponent(
17         JList<? extends Font> list, Font value, int index,
18         boolean isSelected, boolean cellHasFocus)
19     {
20         font = value;
21         background = isSelected ? list.getSelectionBackground() :
22                         list.getBackground();
23         foreground = isSelected ? list.getSelectionForeground() :
24                           list.getForeground();
25         return this;
26     }
27
28     public void paintComponent(Graphics g)
29     {
30         String text = font.getFamily();
31         FontMetrics fm = g.getFontMetrics(font);
32         g.setColor(background);
33         g.fillRect(0, 0, getWidth(), getHeight());
34         g.setColor(foreground);
35         g.setFont(font);
36         g.drawString(text, 0, fm.getAscent());
37     }
38 }
```

```

39  public Dimension getPreferredSize()
40  {
41      String text = font.getFamily();
42      Graphics g = getGraphics();
43      FontMetrics fm = g.getFontMetrics(font);
44      return new Dimension(fm.stringWidth(text), fm.getHeight());
45  }
46 }

```

javax.swing.JList<E> 1.2

- **Color getBackground()**
Возвращает цвет фона для ячеек, невыбранных из списка.
- **Color getSelectionBackground()**
Возвращает цвет фона для ячеек, выбранных из списка.
- **Color getForeground()**
Возвращает цвет переднего плана для ячеек, невыбранных из списка.
- **Color getSelectionForeground()**
Возвращает цвет переднего плана для ячеек, выбранных из списка.
- **void setCellRenderer(ListCellRenderer<? super E> cellRenderer)**
Устанавливает средство для воспроизведения ячеек из списка.

javax.swing.ListCellRenderer<E> 1.2

- **Component getListCellRendererComponent(JList<? extends E> list, E item, int index, boolean isSelected, boolean hasFocus)**
Возвращает компонент, метод **paint()** которого воспроизводит содержимое ячеек из списка. Если размеры ячеек не фиксированы, этот компонент должен также реализовать метод **getPreferredSize()**.

Параметры:	list	Список, ячейки которого воспроизводятся
	item	Воспроизводимый элемент списка
	index	Индекс, обозначающий место для хранения элемента в модели
	isSelected	Принимает логическое значение true , если указанная ячейка выбрана
	hasFocus	Принимает логическое значение true , если указанная ячейка обладает фокусом ввода

10.2. Таблицы

Компонент **JTable** служит для отображения таблицы в виде двухмерной сетки объектов. Таблицы широко применяются при построении ГПИ, поэтому

разработчики библиотеки Swing уделили должное внимание созданию данного компонента. Компонент `JTable` имеет очень сложную структуру, но она, в отличие от других компонентов Swing, практически скрыта от пользователей. С помощью всего нескольких строк кода можно создать полноценную таблицу. Но для составления специальных таблиц придется написать дополнительный код, оформить их особый внешний вид и задать конкретное их поведение в разрабатываемой прикладной программе.

В этом разделе поясняется, каким образом создаются простые таблицы, как организуется взаимодействие пользователей с ними и как они чаще всего настраиваются. Аналогично другим сложным компонентам Swing, все особенности манипулирования таблицами невозможно подробно описать в одном разделе. Поэтому для углубленного изучения данного вопроса рекомендуется следующая дополнительная литература: *Graphic Java™: Mastering the JFC, Volume II: Swing, 3rd Edition* Дэвида М. Гери (David M. Geary; издательство Prentice Hall, 1999 г.) или *Core Swing* Кима Топли (Kim Topley; издательство Prentice Hall, 1999 г.).

10.2.1. Простая таблица

Подобно компоненту `JList`, компонент `JTable` не хранит свои данные, а получает их из *модели таблицы*. Класс `JTable` содержит конструктор, который заключает двухмерный массив объектов в оболочку модели, используемой по умолчанию. Именно такой способ применяется в рассматриваемом здесь первом примере создания таблиц. Модели таблиц более подробно обсуждаются далее в этой главе.

На рис. 10.5 приведена типичная таблица с описаниями свойств планет Солнечной системы. Следует иметь в виду, что в столбце `Gaseous` указывается логическое значение `true`, если планета состоит в основном из водорода и гелия, а иначе — логическое значение `false`. Значения в столбце `Color` пока еще не имеют какого-то определенного смысла, но этот столбец понадобится в следующих примерах.

Planet	Radius	Moons	Gaseous	Color
Mercury	2440.0	0	false	java.awt.C...
Venus	6052.0	0	false	java.awt.C...
Earth	6378.0	11	false	java.awt.C...
Mars	3397.0	2	false	java.awt.C...
Jupiter	71492.0	16	true	java.awt.C...
Saturn	60268.0	18	true	java.awt.C...
Uranus	25559.0	17	true	java.awt.C...
Neptune	24766.0	8	true	java.awt.C...

Рис. 10.5. Пример простой таблицы

Как следует из приведенного ниже фрагмента кода, взятого из листинга 10.5, где представлен исходный код рассматриваемого здесь примера программы, данные таблицы хранятся в виде двухмерного массива значений типа `Object`.

```
Object[][] cells =
{
    { "Mercury", 2440.0, 0, false, Color.YELLOW },
    { "Venus", 6052.0, 0, false, Color.YELLOW },
    . . .
}
```

 **На заметку!** В данном случае используются преимущества автоматического преобразования примитивных типов. Так, значения во втором, в третьем и четвертом столбцах преобразуются в объекты типа `Double`, `Integer` и `Boolean`.

Для отображения каждого объекта таблица вызывает метод `toString()`. Поэтому цвета в последнем столбце представлены в виде строк `java.awt.Color[r=..., g=..., b=...]`.

Сначала имена столбцов задаются в отдельном массиве символьных строк следующим образом:

```
String[] columnNames = { "Planet", "Radius", "Moons",
                        "Gaseous", "Color" };
```

Затем из массивов ячеек и имен столбцов составляется таблица, как показано ниже.

```
JTable table = new JTable(cells, columnNames);
```

 **На заметку!** Следует иметь в виду, что компонент `JTable`, в отличие от компонента `JList`, не является обобщенным. И для этого имеются веские основания. Если в списке предполагается наличие однотипных элементов, то в таблице зачастую находятся разнотипные элементы. В рассматриваемом здесь примере названия планет представлены символьными строками, цвет — строками типа `java.awt.Color[r=..., g=..., b=...]` и т.д.

Таблицу можно также снабдить полосами прокрутки, заключив ее в оболочку компонента `JScrollPane`, как показано ниже. При прокрутке таблицы ее заголовок не исчезает из виду.

```
JScrollPane pane = new JScrollPane(table);
```

Если щелкнуть кнопкой мыши на одном из столбцов и перетащить его влево или вправо, весь столбец визуально отделяется от других столбцов небольшим промежутком (рис. 10.6). Перетаскиваемый столбец можно опустить на любом другом месте. Такое переупорядочение столбцов возможно только в представлении таблицы и никак не влияет на модель данных.

Чтобы изменить размеры столбца, достаточно навести курсор на границу раздела двух столбцов. Как только указатель мыши примет вид двойной стрелки, останется лишь перетащить границу раздела столбцов в нужное место (рис. 10.7).

Чтобы выбрать строку в таблице, достаточно щелкнуть на любом месте в этой строке. Выбранные строки таблицы выделяются другим цветом. Далее будет показано, каким образом организуется обработка событий выбора из таблицы. В рассматриваемом здесь примере допускается редактирование ячеек таблицы,

но внесенные в них изменения не сохраняются. Поэтому при разработке собственной прикладной программы вам придется выбирать одно из двух: вообще запретить редактирование в ячейках таблицы или организовать обработку событий редактирования и обновление модели таблицы. Подробнее об этом — далее в этом разделе.

Planet	Radius	Moons	Gaseous	Color
Mercury	2440.0		false	java.awt.C...
Venus	6052.0		false	java.awt.C...
Earth	6378.0		false	java.awt.C...
Mars	3397.0		false	java.awt.C...
Jupiter	71492.0		true	java.awt.C...
Saturn	60268.0		true	java.awt.C...
Uranus	25559.0		true	java.awt.C...
Neptune	24766.0		true	java.awt.C...

Рис. 10.6. Перемещение столбца в таблице

Planet	Radius	Moons	Gas	Color
Mercury	2440.0	0	false	java.awt.Color[r=...
Venus	6052.0	0	false	java.awt.Color[r=...
Earth	6378.0	1	false	java.awt.Color[r=...
Mars	3397.0	2	false	java.awt.Color[r=...
Jupiter	71492.0	16	true	java.awt.Color[r=...
Saturn	60268.0	18	true	java.awt.Color[r=...
Uranus	25559.0	17	true	java.awt.Color[r=...
Neptune	24766.0	8	true	java.awt.Color[r=...

Рис. 10.7. Изменение размеров столбца в таблице

И наконец, если щелкнуть на заголовке столбца, строки таблицы будут автоматически отсортированы. Если щелкнуть еще раз, сортировка будет произведена в обратном порядке. Такое поведение активизируется при вызове следующего метода:

```
able.setAutoCreateRowSorter(true);
```

Для вывода таблицы на печать вызывается приведенный ниже метод. В итоге появляется диалоговое окно печати, с помощью которого таблица выводится на печатающее устройство. Параметры вывода на печать, доступные в этом окне, рассматриваются в главе 11.

```
table.print()
```



На заметку! Если в рассматриваемом здесь примере программы изменить фрейм из класса **TableTest** таким образом, чтобы по высоте он был выше таблицы, то под таблицей появится серая область. В отличие от компонентов **JList** и **JTree**, таблица, составляемая с помощью компонента **JTable**, не заполняет область просмотра на панели прокрутки. И это может вызвать определенные затруднения, если требуется организовать поддержку перетаскивания. (Подробнее об этом речь пойдет в главе 11.) В таком случае придется вызвать следующий метод:

```
table.setFillsViewportHeight(true);
```



Внимание! Если таблица не вписывается в панель прокрутки, ее заголовок придется добавить явным образом, сделав следующий вызов:

```
add(table.getTableHeader(), BorderLayout.NORTH);
```

Листинг 10.5. Исходный код из файла table/TableTest.java

```

1 package table;
2
3 import java.awt.*;
4 import java.awt.print.*;
5
6 import javax.swing.*;
7
8 /**
9  * В этой программе демонстрируется порядок
10 * отображения простой таблицы
11 * @version 1.13 2016-05-10
12 * @author Cay Horstmann
13 */
14 public class TableTest
15 {
16     public static void main(String[] args)
17     {
18         EventQueue.invokeLater(() ->
19         {
20             JFrame frame = new PlanetTableFrame();
21             frame.setTitle("TableTest");
22             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23             frame.setVisible(true);
24         });
25     }
26 }
27
28 /**
29  * Этот фрейм содержит таблицу с данными о планетах
30  */
31 class PlanetTableFrame extends JFrame
32 {
33     private String[] columnNames = { "Planet", "Radius", "Moons",
34                                     "Gaseous", "Color" };
35     private Object[][] cells = {
36         { "Mercury", 2440.0, 0, false, Color.YELLOW },
37         { "Venus", 6052.0, 0, false, Color.YELLOW },
38         { "Earth", 6378.0, 1, false, Color.BLUE },

```

```

39         { "Mars", 3397.0, 2, false, Color.RED },
40         { "Jupiter", 71492.0, 16, true, Color.ORANGE },
41         { "Saturn", 60268.0, 18, true, Color.ORANGE },
42         { "Uranus", 25559.0, 17, true, Color.BLUE },
43         { "Neptune", 24766.0, 8, true, Color.BLUE },
44         { "Pluto", 1137.0, 1, false, Color.BLACK } ;
45
46     public PlanetTableFrame()
47     {
48         final JTable table = new JTable(cells, columnNames);
49         table.setAutoCreateRowSorter(true);
50         add(new JScrollPane(table), BorderLayout.CENTER);
51         JButton printButton = new JButton("Print");
52         printButton.addActionListener(event ->
53         {
54             try { table.print(); }
55             catch (SecurityException | PrinterException ex)
56             { ex.printStackTrace(); }
57         });
58         JPanel buttonPanel = new JPanel();
59         buttonPanel.add(printButton);
60         add(buttonPanel, BorderLayout.SOUTH);
61         pack();
62     }
63 }

```

javax.swing.JTable 1.2

- **JTable(Object[][] entries, Object[] columnNames)**

Составляет таблицу с моделью, используемой по умолчанию.

- **void print() 5.0**

Отображает диалоговое окно печати и выводит таблицу на печать.

- **boolean getAutoCreateRowSorter() 6**

- **void setAutoCreateRowSorter(boolean newValue) 6**

Получают или устанавливают свойство **autoCreateRowSorter**. По умолчанию оно принимает логическое значение **false**. Если установлено именно это логическое значение, то при любом изменении модели таблицы будет автоматически устанавливаться сортировщик строк, выбираемый по умолчанию.

- **boolean getFillsViewportHeight() 6**

- **void setFillsViewportHeight(boolean newValue) 6**

Получают или устанавливают свойство **fillsViewportHeight**. По умолчанию оно принимает логическое значение **false**. Если установлено именно это логическое значение, то таблица всегда будет заполнять объемлющую область просмотра.

10.2.2. Модели таблиц

Описанные в предыдущем разделе данные таблицы хранились в виде двухмерного массива. Но такой способ в прикладном коде обычно не применяется. Вместо вывода данных в массив для их отображения в табличном виде рекомендуется реализовать собственную модель таблицы.

Реализовать модель таблицы совсем не трудно, поскольку для этой цели предусмотрен отдельный класс `AbstractTableModel` с большинством требующихся методов. Остается лишь предоставить следующие три метода:

```
public int getRowCount();
public int getColumnCount();
public Object getValueAt(int row, int column);
```

Метод `getValueAt()` можно реализовать несколькими способами. Так, если требуется отобразить набор строк типа `RowSet`, содержащий результат запроса к базе данных, то для этого подойдет следующий вариант реализации данного метода:

```
public Object getValueAt(int r, int c)
{
    try
    {
        rowSet.absolute(r + 1);
        return rowSet.getObject(c + 1);
    }
    catch (SQLException e)
    {
        e.printStackTrace();
        return null;
    }
}
```

Следующий пример программы еще проще. В ней составляется таблица из рассчитанных значений, а именно: роста инвестиций при разных учетных ставках (рис. 10.8).

	5%	6%	7%	8%	9%	10%
100000.00	100000.00	100000.00	100000.00	100000.00	100000.00	100000.00
105000.00	105000.00	107000.00	108000.00	109000.00	110000.00	111000.00
110250.00	112360.00	114490.00	116640.00	118810.00	121000.00	123100.00
115762.50	119101.60	122504.30	125971.20	129502.90	133100.00	136715.61
121550.63	125247.70	131079.60	136048.90	141158.16	146410.00	151051.00
127628.16	133822.56	140255.17	146932.81	153862.40	160882.16	167710.01
134009.56	141851.91	150073.04	158687.43	167710.01	177156.10	187156.10
140710.04	150363.03	160578.15	171382.43	182803.91	194871.71	207156.10
147745.54	159384.81	171818.62	185093.02	199256.26	214358.88	230794.77
155132.82	168947.90	183845.92	199900.46	217189.33	235794.77	259374.25
162889.46	179084.77	196715.14	215892.50	236736.37	259374.25	285311.67
171033.94	189829.86	210485.20	233163.90	258042.64	285311.67	313842.84
179585.63	201219.65	225219.16	251817.01	281266.48	313842.84	345227.12
188564.91	213292.83	240984.50	271962.37	305580.46	345227.12	380558.46
197993.16	226090.40	257853.42	293719.36	334172.70	379749.83	417724.82
207892.82	239655.82	275903.15	317216.91	364248.25	417724.82	464248.25

Рис. 10.8. Пример таблицы с данными о росте инвестиций при разных учетных ставках

В данном примере метод `getValueAt()` выполняет расчет соответствующего значения и форматирует его, как показано ниже.

```
public Object getValueAt(int r, int c)
{
    double rate = (c + minRate) / 100.0;
    int nperiods = r;
```

```
double futureBalance =
    INITIAL_BALANCE * Math.pow(1 + rate, nperiods);
return String.format("%.2f", futureBalance);
}
```

А методы `getRowCount()` и `getColumnCount()` просто возвращают количество строк и столбцов соответственно:

```
public int getRowCount() { return years; }
public int getColumnCount() { return maxRate - minRate + 1; }
```

Если заголовки столбцов не заданы явно, то для них в методе `getColumnName()` из класса `AbstractTableModel` по умолчанию используются имена A, B, C и т.д. Для изменения этих имен следует переопределить метод `getColumnName()`. В данном примере в качестве заголовка столбца используется учетная ставка, как показано ниже. Весь исходный код программы из данного примера представлен в листинге 10.6.

```
public String getColumnName(int c) { return (c + minRate) + "%"; }
```

Листинг 10.6. Исходный код из файла `tableModel/InvestmentTable.java`

```
1 package tableModel;
2
3 import java.awt.*;
4
5 import javax.swing.*;
6 import javax.swing.table.*;
7
8 /**
9  * В этой программе демонстрируется построение таблицы по ее модели
10 * @version 1.03 2016-05-10
11 * @author Cay Horstmann
12 */
13 public class InvestmentTable
14 {
15     public static void main(String[] args)
16     {
17         EventQueue.invokeLater(() ->
18         {
19             JFrame frame = new InvestmentTableFrame();
20             frame.setTitle("InvestmentTable");
21             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22             frame.setVisible(true);
23         });
24     }
25 }
26
27 /**
28  * Этот фрейм содержит таблицу капиталовложений
29  */
30 class InvestmentTableFrame extends JFrame
31 {
32     public InvestmentTableFrame()
33     {
34         TableModel model = new InvestmentTableModel(30, 5, 10);
35         JTable table = new JTable(model);
```

```
36     add(new JScrollPane(table));
37     pack();
38 }
39 }
40 }
41 /**
42 * В этой модели таблицы рассчитывается содержимое ячеек таблицы
43 * всякий раз, когда оно запрашивается. В таблице представлен рост
44 * капиталовложений в течение ряда лет при разных учетных ставках
45 */
46 class InvestmentTableModel extends AbstractTableModel
47 {
48     private static double INITIAL_BALANCE = 100000.0;
49
50     private int years;
51     private int minRate;
52     private int maxRate;
53
54 /**
55 * Конструирует модель таблицы капиталовложений
56 * @param y Количество лет
57 * @param r1 Наинизшая учетная ставка для составления таблицы
58 * @param r2 Наивысшая учетная ставка для составления таблицы
59 */
60     public InvestmentTableModel(int y, int r1, int r2)
61     {
62         years = y;
63         minRate = r1;
64         maxRate = r2;
65     }
66
67     public int getRowCount()
68     {
69         return years;
70     }
71
72     public int getColumnCount()
73     {
74         return maxRate - minRate + 1;
75     }
76
77     public Object getValueAt(int r, int c)
78     {
79         double rate = (c + minRate) / 100.0;
80         int nperiods = r;
81         double futureBalance =
82             INITIAL_BALANCE * Math.pow(1 + rate, nperiods);
83         return String.format("%.2f", futureBalance);
84     }
85
86     public String getColumnName(int c)
87     {
88         return (c + minRate) + "%";
89     }
90 }
```

javax.swing.table.TableModel 1.2

- **int getRowCount()**
- **int getColumnCount()**
Получают количество строк и столбцов в модели таблицы.
- **Object getValueAt(int row, int column)**
Получает значение из указанных строки и столбца.
- **void setValueAt(Object newValue, int row, int column)**
Устанавливает новое значение в указанных строке и столбце.
- **boolean isCellEditable(int row, int column)**
Возвращает логическое значение **true**, если ячейка в указанных строке и столбце доступна для редактирования.
- **String getColumnName(int column)**
Получает заголовок столбца.

10.2.3. Манипулирование строками и столбцами таблицы

В этом разделе рассматриваются способы манипулирования строками и столбцами таблицы. Таблица, составленная средствами Swing, имеет несимметричную структуру, допускающую выполнение разных операций над строками и столбцами. Дело в том, что компонент **JTable**, реализующий таблицу в библиотеке Swing, предназначен для отображения строк одинаковой структуры, например, данных из таблиц базы данных, а не объектов в виде произвольной двухмерной сетки. Такая асимметрия демонстрируется в данном разделе на конкретных примерах.

10.2.3.1. Классы столбцов таблицы

В следующем примере снова рассматривается таблица с данными о планетах, но теперь особое внимание уделяется типам данных в столбцах. В частности, приведенный ниже метод из модели таблицы возвращает класс с описанием типа столбца. Эти сведения используются в классе **JTable** для выбора средства воспроизведения, подходящего для конкретного класса столбца.

```
Class<?> getColumnClass(int columnIndex)
```

В табл. 10.1 перечислены действия, выполняемые по умолчанию, для воспроизведения столбцов по их типам (и соответствующим классам). Так, в столбцах таблицы на рис. 10.9 показаны флагки и изображения. Автор выражает искреннюю благодарность Джиму Эвинсу (Jim Evins) за любезно предоставленные изображения планет. Чтобы воспроизвести в таблице столбцы других типов, следует установить специальное средство воспроизведения, как поясняется далее, в разделе 10.2.4.

Таблица 10.1. Действия, выполняемые по умолчанию, для воспроизведения столбцов по их типам

Тип	Как воспроизводится
Boolean	Флажок
Icon	Изображение
Object	Символьная строка

Rows	US	Moons	Gaseous	Color	Image
Earth	6,378	1	<input type="checkbox"/>	java.awt.Color[r=255,g=0,b=255]	
Mars	3,397	2	<input type="checkbox"/>	java.awt.Color[r=255,g=0,b=0]	
Jupiter	71,492	16	<input checked="" type="checkbox"/>	java.awt.Color[r=255,g=200,b=0]	
Saturn	60,268	18	<input checked="" type="checkbox"/>	java.awt.Color[r=255,g=200,b=0]	

Рис. 10.9. Пример таблицы с данными о планетах, в столбцах которой воспроизводятся флагки и изображения планет

10.2.3.2. Доступ к столбцам таблицы

Компонент `JTable` сохраняет сведения обо всех столбцах таблицы в объектах типа `TableColumn`, а объект типа `TableColumnModel` манипулирует столбцами. (На рис. 10.1 Осемнадцати представлены отношения между наиболее важными классами таблиц.) Если столбцы таблицы не предполагается перемещать или вставлять динамически, то обращаться к модели столбцов таблицы (и соответствующему классу) придется нечасто. А чаще всего обращаться к этой модели требуется для получения объекта типа `TableColumn`, как показано ниже.

```
int columnIndex = . . . ;
TableColumn column = table.getColumnModel().getColumn(columnIndex);
```

10.2.3.3. Изменение размеров столбцов таблицы

Класс `TableColumn` позволяет изменять размеры столбцов. В частности, для указания предпочтительной, минимальной и максимальной ширины столбца используются следующие методы:

```
void setPreferredWidth(int width)
void setMinWidth(int width)
void setMaxWidth(int width)
```

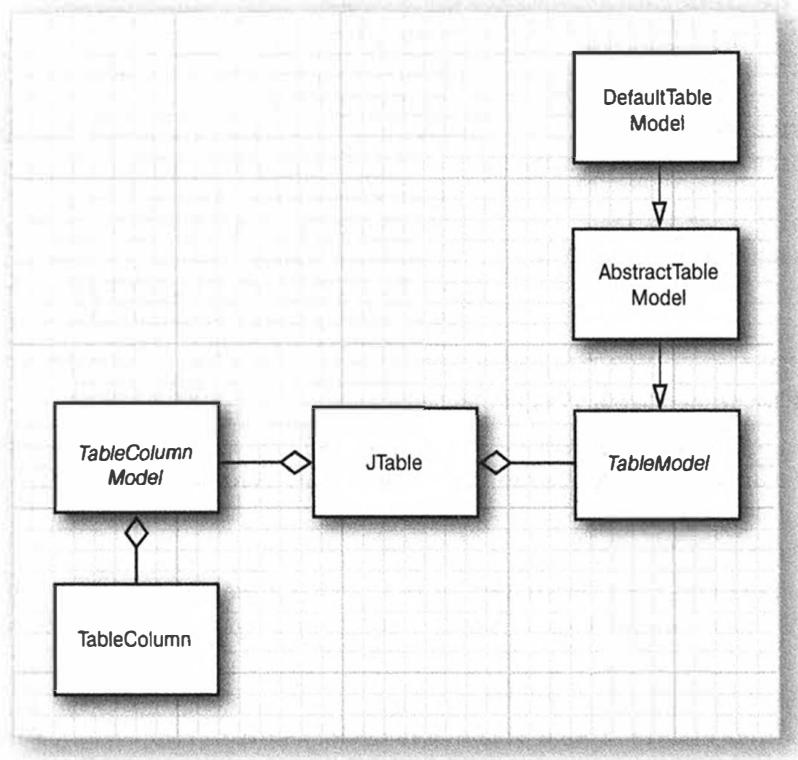


Рис. 10.10. Отношения между табличными классами

Полученная в итоге информация используется компонентом составления таблиц для размещения столбцов. Чтобы разрешить или запретить пользователю изменять размеры столбцов, достаточно вызвать метод `void setResizable(boolean resizable)`, а для того чтобы изменить размеры столбца программным путем — метод `void setWidth(int width)`.

Если изменяются размеры одного столбца, то по умолчанию общие размеры таблицы остаются прежними. Это, конечно, приводит к увеличению или уменьшению размеров остальных столбцов. По умолчанию подобные изменения распространяются на все столбцы, расположенные справа от того столбца, размеры которого изменяются. Благодаря этому требующиеся размеры всех столбцов можно указывать слева направо.

В табл. 10.2 перечислены режимы изменения размеров таблицы при изменении размеров столбца. Их можно указать с помощью метода `void setAutoResizeMode(int mode)` из класса `JTable`.

Таблица 10.2. Режимы изменения размеров таблицы

Режим	Поведение
<code>AUTO_RESIZE_OFF</code>	Размеры остальных столбцов остаются прежними, а размеры таблицы изменяются
<code>AUTO_RESIZE_NEXT_COLUMN</code>	Изменяются только размеры следующего столбца
<code>AUTO_RESIZE_SUBSEQUENT_COLUMNS</code>	Равномерно изменяются размеры всех последующих столбцов (выбирается по умолчанию)
<code>AUTO_RESIZE_LAST_COLUMN</code>	Изменяются размеры только последнего столбца
<code>AUTO_RESIZE_ALL_COLUMNS</code>	Размеры всех столбцов таблицы изменяются равномерно (это не самый лучший вариант, так как он не дает пользователю возможность подгонять многие столбцы под требующиеся размеры)

10.2.3.4. Изменение размеров строк таблицы

Изменение высоты строк таблицы производится непосредственно в классе `JTable`. Если высота ячеек выше заданной по умолчанию, то ее придется указать явно, как показано ниже.

```
table.setRowHeight(height);
```

По умолчанию все строки таблицы имеют одинаковую высоту, но данную установку можно изменить с помощью метода `setRowHeight()` следующим образом:

```
table.setRowHeight(row, height);
```

Конкретная высота строки задается перечисленными выше методами за вычетом величины междустрочного интервала. По умолчанию величина этого интервала равна 1, но любое другое его значение можно указать следующим образом:

```
table.setRowMargin(margin);
```

10.2.3.5. Выбор строк, столбцов и ячеек таблицы

В зависимости от заданного режима пользователь может выбирать строки, столбцы и отдельные ячейки таблицы. По умолчанию допускается выбор строки, т.е. после щелчка кнопкой мыши на одной из ячеек будет выбрана вся строка, как было показано на рис. 10.9. Для отмены режима выбора строк достаточно вызвать следующий метод:

```
table.setRowSelectionAllowed(false)
```

В режиме выбора строк пользователю можно разрешить выбор только одной строки, нескольких смежных или несмежных строк. Для этого необходимо получить модель выбора и вызвать ее метод `setSelectionMode()` следующим образом:

```
table.getSelectionModel().setSelectionMode(mode);
```

В качестве параметра `mode` можно указать следующие значения:

```
ListSelectionModel.SINGLE_SELECTION
ListSelectionModel.SINGLE_INTERVAL_SELECTION
ListSelectionModel.MULTIPLE_INTERVAL_SELECTION
```

По умолчанию режим выбора столбцов отключен. Его можно включить, вызвав следующий метод:

```
table.setColumnSelectionAllowed(true)
```

Одновременное включение режимов выбора строк и столбцов равнозначно включению режима выбора ячеек (рис. 10.11). Эти режимы можно указать явно, вызвав следующий метод:

```
table.setCellSelectionEnabled(true)
```

Planet	Radius	Moons	Gaseous	Color	Image
Mars	3,397	2	<input type="checkbox"/>	java.awt.Color[r=255,g=0,b=0]	
Jupiter	71,492	16	<input checked="" type="checkbox"/>	java.awt.Color[r=255,g=200,b=0]	
Saturn	60,268	18	<input checked="" type="checkbox"/>	java.awt.Color[r=255,g=200,b=0]	
Uranus	25,559	17	<input checked="" type="checkbox"/>	java.awt.Color[r=0,g=0,b=255]	

Рис. 10.11. Выбор ряда ячеек

Чтобы посмотреть, каким образом выбор ячеек происходит на практике, запустите на выполнение пример программы, исходный код которой представлен в листинге 10.7. Активизируйте режим выбора строки, столбца или ячейки из меню Selection (Выбор) и понаблюдайте за тем, как изменяется поведение таблицы в данном режиме.

С помощью методов `getSelectedRows()` и `getSelectedColumns()` можно выяснить, какие именно строки и столбцы были выбраны. Оба метода возвращают массив `int[]` индексов выбранных элементов. Следует, однако, иметь в виду, что значения индексов берутся из представления таблицы, а не из базовой

модели таблицы. Попытайтесь выбрать строки и столбцы, а затем перетащите столбцы в разные места и отсортируйте строки, щелкая на заголовках столбцов. Выбрав из меню пункт Print Selection (Выбор для печати), посмотрите, какие именно строки и столбцы отображаются как выбранные. Если же требуется преобразовать значения индексов таблицы в значения индексов модели таблицы, то для этой цели можно воспользоваться методами `convertRowIndexToModel()` и `convertColumnIndexToModel()` из класса `JTable`.

10.2.3.6. Сортировка строк таблицы

Как было показано в рассмотренном выше первом примере таблицы, компонент `JTable` можно без особого труда дополнить функцией сортировки строк таблицы, вызвав метод `setAutoCreateRowSorter()`. Но для того чтобы получить больше возможностей для управления сортировкой, в компоненте `JTable` следует установить и настроить объект типа `TableRowSorter<M>`. Параметр типа `M` обозначает тип модели; им должен быть подтип интерфейса `TableModel`:

```
TableRowSorter<TableModel> sorter =
    new TableRowSorter<TableModel>(model);

table.setRowSorter(sorter);
```

Некоторые столбцы должны быть исключены из сортировки (например, столбец с изображениями в таблице с данными о планетах). Чтобы исключить отдельные столбцы таблицы из сортировки, достаточно вызвать следующий метод:

```
sorter.setSortable(IMAGE_COLUMN, false);
```

Для каждого столбца можно также установить специальное средство сравнения. В рассматриваемом здесь примере предполагается отсортировать цвета в столбце `Color`, отдавая предпочтение синему и зеленому цвету над красным. Если щелкнуть на столбце `Color`, планеты синего цвета окажутся внизу таблицы. Такой результат сортировки по цвету достигается с помощью следующего метода:

```
sorter.setComparator(COLOR_COLUMN, new Comparator<Color>()
{
    public int compare(Color c1, Color c2)
    {
        int d = c1.getBlue() - c2.getBlue();
        if (d != 0) return d;
        d = c1.getGreen() - c2.getGreen();
        if (d != 0) return d;
        return c1.getRed() - c2.getRed();
    }
});
```

Если не указать компаратор для столбцов, сортировка будет произведена в следующем порядке.

1. Если класс столбца относится к типу `String`, то по умолчанию используется средство сортировки, возвращаемое из метода `Collator.getInstance()`. Это средство сортирует символьные строки в соответствии с текущими региональными настройками. (Подробнее о региональных настройках и средствах сортировки см. в главе 7.)

2. Если класс столбца реализует интерфейс Comparable, то используется метод compareTo().
3. Если для сортировки установлен преобразователь строк таблицы типа TableStringConverter, то сортировку символьных строк, возвращаемых методом toString() этого преобразователя, необходимо выполнять с помощью средства сортировки, выбирамого по умолчанию. Чтобы воспользоваться именно таким способом сортировки, необходимо определить преобразователь строк таблицы следующим образом:

```
sorter.setStringConverter(new TableStringConverter()
{
    public String toString(TableModel model, int row, int column)
    {
        Object value = model.getValueAt(row, column);
        преобразовать объект value в символьную строку и возвратить ее
    }
});
```

4. В противном случае вызывается метод toString() со значениями в ячейках, которые упорядочиваются выбираемым по умолчанию средством сортировки.

10.2.3.7. Фильтрация строк таблицы

Помимо сортировки строк таблицы, класс TableRowSorter позволяет избирательно скрывать строки. Этот процесс называется *фильтрацией*. Чтобы активизировать режим фильтрации, следует установить соответствующий фильтр типа RowFilter. Например, для того чтобы отобрать все строки таблицы, содержащие хотя бы один спутник планеты, достаточно вызвать следующий метод:

```
sorter.setRowFilter(RowFilter.numberFilter(
    ComparisonType.NOT_EQUAL, 0, MOONS_COLUMN));
```

Здесь используется предопределенный фильтр чисел. Чтобы создать такой фильтр, понадобятся следующие средства.

- Вид сравнения (одна из констант EQUAL, NOT_EQUAL, AFTER или BEFORE).
- Объект подкласса, производного от класса Number (например, Integer или Double). Допускаются только те объекты, которые имеют тот же класс, что и у данного объекта типа Number.
- Ни одного значения или несколько значений индекса столбца. Если эти значения не заданы, поиск будет производиться по всем столбцам.

Аналогичным образом в статическом методе RowFilter.dateFilter() создается фильтр дат. Отличие заключается лишь в том, что вместо объекта типа Number задается объект типа Date. И наконец, в статическом методе RowFilter.regexFilter() создается фильтр, осуществляющий поиск символьных строк, совпадающих с регулярным выражением. Например, в следующей строке кода отбираются только те планеты, название которых не оканчивается на "з". (Подробнее о регулярных выражениях см. в главе 2.)

```
sorter.setRowFilter(RowFilter.regexFilter(".*[^\z]$",
    PLANET_COLUMN));
```

Кроме того, фильтры можно применять в различных сочетаниях с помощью методов `andFilter()`, `orFilter()` и `notFilter()`. Так, если требуется отобрать планеты, названия которых не оканчиваются на "s" и которые имеют как минимум один спутник, для этого можно применить фильтры в следующем сочетании:

```
sorter.setRowFilter(RowFilter.andFilter(Arrays.asList(
    RowFilter.regexFilter(".*[^s]$", PLANET_COLUMN),
    RowFilter.numberFilter(
        ComparisonType.NOT_EQUAL, 0, MOONS_COLUMN))));
```



Внимание! Обратите внимание на то, что в методах `andFilter()` и `orFilter()` употребляется не переменное число аргументов, а единственный параметр типа `Iterable`.

Чтобы реализовать собственный фильтр, необходимо предоставить объект подкласса, производного от класса `RowFilter`, и реализовать метод `include()` с целью указать те строки таблицы, которые требуется отобрать и отобразить. Сделать это нетрудно, хотя и не так просто в силу обобщенного характера класса `RowFilter`.

У класса `RowFilter<M, I>` имеются два параметра типа, обозначающие типы модели и идентификатора строк таблицы. При манипулировании таблицами модель всегда относится к подтипу `TableModel`, а идентификатор — к типу `Integer`. (Когда-нибудь остальные компоненты будут также поддерживать фильтрацию строк, как в таблицах. Например, для отбора строк в компоненте `JTree` может потребоваться класс `RowFilter<TreeModel, TreePath>`.)

Фильтр строк таблицы должен реализовывать следующий метод:

```
public boolean include(
    RowFilter.Entry<? extends M, ? extends I> entry)
```

В классе `RowFilter` предоставляются методы для получения модели, идентификатора строк таблицы и значения по заданному индексу. Таким образом, фильтрацию можно производить как по идентификатору строк таблицы, так и по их содержимому. Например, с помощью следующего фильтра отображается каждая вторая строка таблицы:

```
RowFilter<TableModel, Integer> filter =
    new RowFilter<TableModel, Integer>()
{
    public boolean include(
        Entry<? extends TableModel, ? extends Integer> entry)
    {
        return entry.getValue(MOONS_COLUMN) % 2 == 0;
    }
};
```

Если же требуется отобрать только те планеты, которые содержат четное количество спутников, то вместо приведенного выше фильтра можно попытаться применить следующий фильтр:

```
((Integer) entry.getValue(MOONS_COLUMN)) % 2 == 0
```

В рассматриваемом здесь примере программе пользователю разрешается скрывать произвольные строки таблицы. Индексы скрытых строк сохраняются в наборе строк. А в фильтр строк включаются все строки таблицы, индекс которых отсутствует в данном наборе.

Такой механизм фильтрации не предназначен для применения фильтров, критерии которых изменяются с течением времени. В рассматриваемом здесь примере программы вызов приведенного ниже метода повторяется всякий раз, когда изменяется набор скрытых строк таблицы. Фильтр применяется сразу же, как только он будет установлен.

```
sorter.setRowFilter(filter);
```

10.2.3.8. Скрытие и показ столбцов таблицы

Как было показано в предыдущем подразделе, строки таблицы можно отфильтровывать по их содержимому или идентификатору. А для скрытия столбцов понадобится совершенно другой механизм.

Метод `removeColumn()` из класса `JTable` позволяет удалить столбец, определяемый параметром `TableColumn`, из представления таблицы, т.е. скрыть его от пользователя, оставив в составе модели. Ниже показано, каким образом конкретный объект, описывающий столбец таблицы, извлекается из модели таблицы по известному номеру, получаемому, например, с помощью метода `getSelectedColumns()`.

```
TableColumnModel columnModel = table.getColumnModel();
TableColumn column = columnModel.getColumn(i);
table.removeColumn(column);
```

Если запомнить этот объект, то впоследствии его можно ввести обратно в модель таблицы следующим образом:

```
table.addColumn(column);
```

Этот метод добавляет столбец в конец таблицы. Если же столбец требуется расположить в каком-нибудь другом месте, то для его перемещения на это место следует вызвать метод `moveColumn()`.

Кроме того, создав объект типа `TableColumn`, можно сформировать новый столбец, который соответствует индексу столбца в модели таблицы, как показано ниже. Таким образом, в таблице можно создать несколько столбцов, которые будут представлять один и тот же столбец в модели.

```
table.addColumn(new TableColumn(modelColumnIndex));
```

В рассматриваемом здесь примере программы демонстрируется выбор и фильтрация строк и столбцов таблицы. Исходный код этой программы представлен в листинге 10.7.

Листинг 10.7. Исходный код из файла `tableRowColumn/PlanetTableFrame.java`

```
1 package tableRowColumn;
2
3 import java.awt.*;
4 import java.util.*;
5
6 import javax.swing.*;
7 import javax.swing.table.*;
8
9 /**
10 * Этот фрейм содержит таблицы с данными о планетах
```

```
11  /*
12  public class PlanetTableFrame extends JFrame
13  {
14      private static final int DEFAULT_WIDTH = 600;
15      private static final int DEFAULT_HEIGHT = 500;
16
17      public static final int COLOR_COLUMN = 4;
18      public static final int IMAGE_COLUMN = 5;
19
20      private JTable table;
21      private HashSet<Integer> removedRowIndices;
22      private ArrayList<TableColumn> removedColumns;
23      private JCheckBoxMenuItem rowsItem;
24      private JCheckBoxMenuItem columnsItem;
25      private JCheckBoxMenuItem cellsItem;
26
27      private String[] columnNames = { "Planet", "Radius",
28                                      "Moons", "Gaseous", "Color", "Image" };
29
30      private Object[][][] cells =
31      {
32          { "Mercury", 2440.0, 0, false, Color.YELLOW,
33              new ImageIcon(getClass().getResource("Mercury.gif")) },
34          { "Venus", 6052.0, 0, false, Color.YELLOW,
35              new ImageIcon(getClass().getResource("Venus.gif")) },
36          { "Earth", 6378.0, 1, false, Color.BLUE,
37              new ImageIcon(getClass().getResource("Earth.gif")) },
38          { "Mars", 3397.0, 2, false, Color.RED,
39              new ImageIcon(getClass().getResource("Mars.gif")) },
40          { "Jupiter", 71492.0, 16, true, Color.ORANGE,
41              new ImageIcon(getClass().getResource("Jupiter.gif")) },
42          { "Saturn", 60268.0, 18, true, Color.ORANGE,
43              new ImageIcon(getClass().getResource("Saturn.gif")) },
44          { "Uranus", 25559.0, 17, true, Color.BLUE,
45              new ImageIcon(getClass().getResource("Uranus.gif")) },
46          { "Neptune", 24766.0, 8, true, Color.BLUE,
47              new ImageIcon(getClass().getResource("Neptune.gif")) },
48          { "Pluto", 1137.0, 1, false, Color.BLACK,
49              new ImageIcon(getClass().getResource("Pluto.gif")) } };
50
51      public PlanetTableFrame()
52      {
53          setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
54
55          TableModel model = new DefaultTableModel(cells, columnNames)
56          {
57              public Class<?> getColumnClass(int c)
58              {
59                  return cells[0][c].getClass();
60              }
61          };
62
63          table = new JTable(model);
64
65          table.setRowHeight(100);
66          table.getColumnModel().getColumn(COLOR_COLUMN)
67              .setMinWidth(250);
68          table.getColumnModel().getColumn(IMAGE_COLUMN)
69              .setMinWidth(100);
```

```
69
70     final TableRowSorter<TableModel> sorter =
71         new TableRowSorter<>(model);
72     table.setRowSorter(sorter);
73     sorter.setComparator(COLOR_COLUMN, Comparator
74             .comparing(Color::getBlue)
75             .thenComparing(Color::getGreen)
76             .thenComparing(Color::getRed));
77     sorter.setSortable(IMAGE_COLUMN, false);
78     add(new JScrollPane(table), BorderLayout.CENTER);
79
80     removedRowIndices = new HashSet<>();
81     removedColumns = new ArrayList<>();
82
83     final RowFilter<TableModel, Integer> filter =
84         new RowFilter<TableModel, Integer>()
85     {
86         public boolean include(Entry<? extends TableModel,
87             ? extends Integer> entry)
88         {
89             return !removedRowIndices.contains(entry.getIdentifier());
90         }
91     };
92
93     // создать меню
94
95     JMenuBar menuBar = new JMenuBar();
96     setJMenuBar(menuBar);
97
98     JMenu selectionMenu = new JMenu("Selection");
99     menuBar.add(selectionMenu);
100
101    rowsItem = new JCheckBoxMenuItem("Rows");
102    columnsItem = new JCheckBoxMenuItem("Columns");
103    cellsItem = new JCheckBoxMenuItem("Cells");
104
105    rowsItem.setSelected(table.getRowSelectionAllowed());
106    columnsItem.setSelected(table.getColumnSelectionAllowed());
107    cellsItem.setSelected(table.getCellSelectionEnabled());
108
109    rowsItem.addActionListener(event ->
110    {
111        table.clearSelection();
112        table.setRowSelectionAllowed(rowsItem.isSelected());
113        updateCheckboxMenuItems();
114    });
115
116    columnsItem.addActionListener(event ->
117    {
118        table.clearSelection();
119        table.setColumnSelectionAllowed(
120            columnsItem.isSelected());
121        updateCheckboxMenuItems();
122    });
123    selectionMenu.add(columnsItem);
124
125    cellsItem.addActionListener(event ->
126    {
```

```
128         table.clearSelection();
129         table.setCellSelectionEnabled(cellsItem.isSelected());
130         updateCheckboxMenuItems();
131     });
132     selectionMenu.add(cellsItem);
133
134     JMenu tableMenu = new JMenu("Edit");
135     menuBar.add(tableMenu);
136
137     JMenuItem hideColumnsItem = new JMenuItem("Hide Columns");
138     hideColumnsItem.addActionListener(event ->
139     {
140         int[] selected = table.getSelectedColumns();
141         TableColumnModel columnModel = table.getColumnModel();
142
143         // удалить столбцы из представления таблицы, начиная с
144         // последнего индекса, но не затрагивая номера столбцов
145
146         for (int i = selected.length - 1; i >= 0; i--)
147         {
148             TableColumn column =
149                 columnModel.getColumn(selected[i]);
150             table.removeColumn(column);
151
152             // сохранить удаленные столбцы для отображения
153
154             removedColumns.add(column);
155         }
156     });
157     tableMenu.add(hideColumnsItem);
158
159     JMenuItem showColumnsItem = new JMenuItem("Show Columns");
160     showColumnsItem.addActionListener(event ->
161     {
162         // восстановить все удаленные столбцы
163         for (TableColumn tc : removedColumns)
164             table.addColumn(tc);
165         removedColumns.clear();
166     });
167     tableMenu.add(showColumnsItem);
168
169     JMenuItem hideRowsItem = new JMenuItem("Hide Rows");
170     hideRowsItem.addActionListener(event ->
171     {
172         int[] selected = table.getSelectedRows();
173         for (int i : selected)
174             removedRowIndices.add(table.convertRowIndexToModel(i));
175         sorter.setRowFilter(filter);
176     });
177     tableMenu.add(hideRowsItem);
178
179     JMenuItem showRowsItem = new JMenuItem("Show Rows");
180     showRowsItem.addActionListener(event ->
181     {
182         removedRowIndices.clear();
183         sorter.setRowFilter(filter);
184     });
185     tableMenu.add(showRowsItem);
```

```

186     JMenuItem printSelectionItem =
187         new JMenuItem("Print Selection");
188     printSelectionItem.addActionListener(event ->
189     {
190         int[] selected = table.getSelectedRows();
191         System.out.println("Selected rows:
192             " + Arrays.toString(selected));
193         selected = table.getSelectedColumns();
194         System.out.println("Selected columns:
195             " + Arrays.toString(selected));
196     });
197     tableMenu.add(printSelectionItem);
198 }
199
200
201 private void updateCheckboxMenuItems()
202 {
203     rowsItem.setSelected(table.getRowSelectionAllowed());
204     columnsItem.setSelected(table.getColumnSelectionAllowed());
205     cellsItem.setSelected(table.getCellSelectionEnabled());
206 }
207 }
```

javax.swing.table.TableModel 1.2

- **Class getColumnClass(int columnIndex)**

Возвращает класс для определения типа значений в указанном столбце. Эти сведения используются для сортировки и воспроизведения значений в указанном столбце.

javax.swing.JTable 1.2

- **TableColumnModel getColumnModel()**

Получает модель столбца, описывающую расположение столбцов в таблице.

- **void setAutoResizeMode(int mode)**

Устанавливает режим автоматического изменения размеров столбцов таблицы.

Параметры:

mode Одно из значений следующих констант:

AUTO_RESIZE_OFF,
AUTO_RESIZE_NEXT_COLUMN,
AUTO_RESIZE_SUBSEQUENT_COLUMNS,
AUTO_RESIZE_LAST_COLUMN,
AUTO_RESIZE_ALL_COLUMNS

- **int getRowMargin()**

- **void setRowMargin(int margin)**

Получают или устанавливают ширину интервала между ячейками смежных строк таблицы.

- **int getRowHeight()**

- **void setRowHeight(int height)**

Получают или устанавливают высоту всех строк таблицы по умолчанию.

javax.swing.JTable 1.2 (окончание)

- **int getRowHeight(int row)**
- **void setRowHeight(int row, int height)**
Получают или устанавливают высоту указанной строки таблицы.
- **ListSelectionModel getSelectionModel()**
Возвращает модель выбора списка. Эта модель требуется для того, чтобы сделать выбор между строкой, столбцом и ячейкой.
- **boolean getRowSelectionAllowed()**
- **void setRowSelectionAllowed(boolean b)**
Получают или устанавливают свойство `rowSelectionAllowed`. Если оно принимает логическое значение `true`, то строки выбираются из таблицы, если щелкнуть на ее ячейках.
- **boolean getColumnSelectionAllowed()**
- **void setColumnSelectionAllowed(boolean b)**
Получают или устанавливают свойство `columnSelectionAllowed`. Если оно принимает логическое значение `true`, то столбцы выбираются из таблицы, если щелкнуть на ее ячейках.
- **boolean getCellSelectionEnabled()**
Возвращает логическое значение `true`, если оба свойства, `rowSelectionAllowed` и `columnSelectionAllowed`, принимают логическое значение `true`.
- **void setCellSelectionEnabled(boolean b)**
Присваивает обоим свойствам, `rowSelectionAllowed` и `columnSelectionAllowed`, значение параметра `b`.
- **void addColumn(TableColumn column)**
Добавляет столбец, становящийся последним в представлении таблицы.
- **void moveColumn(int from, int to)**
Перемещает столбец, находящийся в таблице по индексу `from`, на новое место по индексу `to`. Эта операция затрагивает только представление таблицы.
- **void removeColumn(TableColumn column)**
- Удаляет указанный столбец из представления таблицы.
- **int convertRowIndexToModel(int index)** 6
- **int convertColumnIndexToModel(int index)**
Возвращают индекс строки или столбца в модели по указанному индексу. Значение возвращаемого индекса отличается от значения параметра `index`, если производится сортировка или фильтрация строк таблицы или же если столбцы перемещаются или удаляются из таблицы.
- **void setRowSorter(RowSorter<? extends TableModel> sorter)**
Устанавливает средство сортировки строк таблицы.

javax.swing.table.TableColumnModel 1.2

- **TableColumn getColumn(int index)**
Возвращает объект, описывающий столбец по указанному индексу в представлении таблицы.

javax.swing.table.TableColumn 1.2

- **TableColumn(int modelColumnIndex)**

Создает столбец таблицы для представления столбца в модели таблицы по указанному индексу.

- **void setPreferredWidth(int width)**

- **void setMinWidth(int width)**

- **void setMaxWidth(int width)**

Задают предпочтительную, минимальную или максимальную ширину столбца равной значению параметра *width*.

- **void setWidth(int width)**

Задает конкретную ширину столбца равной значению параметра *width*.

- **void setResizable(boolean b)**

Если параметр *b* принимает логическое значение **true**, то допускается изменять размеры столбца.

javax.swing.ListSelectionModel 1.2

- **void setSelectionMode(int mode)**

Задает режим выбора.

Параметры: *mode*

Одно из значений следующих констант:

SINGLE_SELECTION,

SINGLE_INTERVAL_SELECTION или

MULTIPLE_INTERVAL_SELECTION

javax.swing.DefaultRowSorter<M, I> 6

- **void setComparator(int column, Comparator<?> comparator)**

Устанавливает средство для сравнения со значением в указанном столбце.

- **void setSortable(int column, boolean enabled)**

Разрешает или запрещает сортировку для указанного столбца.

- **void setRowFilter(RowFilter<? super M, ? super I> filter)**

Устанавливает фильтр строк таблицы.

javax.swing.table.TableRowSorter<M extends TableModel> 6

- **void setStringConverter(TableRowStringConverter stringConverter)**

Устанавливает преобразователь для сортировки и фильтрации строк таблицы.

javax.swing.table.TableStringConverter<M extends TableModel> 6

- **abstract String toString(TableModel model, int row, int column)**

Преобразует в символьную строку значение из указанного места в модели таблицы. Этот метод можно переопределить.

javax.swing.RowFilter<M, I> 6

- **boolean include(RowFilter.Entry<? extends M, ? extends I> entry)**

Определяет строки таблицы, которые остались после фильтрации. Этот метод можно переопределить.

- **static <M, I> RowFilter<M, I> numberFilter(RowFilter.ComparisonType type, Number number, int... indices)**

- **static <M, I> RowFilter<M, I> dateFilter(RowFilter.ComparisonType type, Date date, int... indices)**

Возвращают фильтр, включающий строки, которые содержат значения, совпадающие со сравниваемым числом или датой. В качестве вида сравнения можно указать значение одной из следующих констант: **EQUAL**, **NOT_EQUAL**, **AFTER** или **BEFORE**. Если заданы любые индексы столбцов в модели, то поиск будет производиться только в этих столбцах, а иначе — во всех столбцах. Для фильтрации чисел класс, определяющий тип значений в ячейках таблицы, должен совпадать с классом параметра **number**.

- **static <M, I> RowFilter<M, I> regexFilter(String regex, int... indices)**

Возвращает фильтр, включающий строки, которые содержат значения, совпадающие со сравниваемым регулярным выражением. Если заданы любые индексы столбцов в модели, то поиск будет производиться только в этих столбцах, а иначе — во всех столбцах. Следует, однако, иметь в виду, что метод **getStringValue()** из класса **RowFilter.Entry** возвращает совпадшую символьную строку.

- **static <M, I> RowFilter<M, I> andFilter(Iterable<? extends RowFilter<? super M, ? super I>> filters)**

- **static <M, I> RowFilter<M, I> orFilter(Iterable<? extends RowFilter<? super M, ? super I>> filters)**

Возвращают фильтр, включающий записи, входящие во все фильтры или хотя бы в один из фильтров.

- **static <M, I> RowFilter<M, I> notFilter(RowFilter<M, I> filter)**

Возвращает фильтр, включающий записи, не входящие в указанный фильтр.

javax.swing.RowFilter.Entry<M, I> 6

- **I getIdentifier()**

Возвращает идентификатор данной записи в строке таблицы.

- **M getModel()**

Возвращает модель данной записи в строке таблицы.

javax.swing.RowFilter.Entry<M, I> 6 (окончание)**• Object getValue(int index)**

Возвращает значение, хранящееся по указанному индексу в данной строке таблицы.

• int getCount()

Возвращает количество значений, хранящихся в данной строке таблицы.

• String getStringValue(int index)

Возвращает значение, хранящееся по указанному индексу в данной строке таблицы и преобразованное в символьную строку. Средство сортировки строк типа **TableRowSorter** создает записи, для которых в данном методе вызывается преобразователь отсортированных результатов в символьные строки.

10.2.4. Воспроизведение и редактирование ячеек

Как было показано в подразделе 10.2.3.2, тип столбца определяет способ воспроизведения ячеек таблицы. По умолчанию для типов Boolean и Icon представляются средства воспроизведения флагков или изображений. А для всех остальных типов приходится самостоятельно устанавливать специальное средство воспроизведения.

10.2.4.1. Воспроизведение ячеек таблицы

Средства воспроизведения ячеек в таблице подобны упоминавшимся ранее средствам воспроизведения ячеек в списке. Они реализуют интерфейс **TableCellRenderer** с приведенным ниже единственным методом.

```
Component getTableCellRendererComponent(JTable table, Object value,
                                       boolean isSelected, boolean hasFocus, int row, int column)
```

Этот метод вызывается всякий раз, когда требуется снова воспроизвести таблицу. Он возвращает компонент, метод **paint()** которого служит для отображения содержимого ячейки.

В таблице, представленной на рис. 10.12, содержатся ячейки типа **Color**. Средство воспроизведения просто возвращает панель с цветом фона в виде объекта цвета, хранящегося в данной ячейке. Требующийся цвет передается в качестве параметра **value** методу, устанавливающему цвет фона, как показано ниже.

```
class ColorTableCellRenderer extends JPanel implements TableCellRenderer {
    public Component getTableCellRendererComponent(
        JTable table, Object value, boolean isSelected,
        boolean hasFocus, int row, int column)
    {
        setBackground((Color) value);
        if (hasFocus)
            setBorder(UIManager.getBorder("Table.focusCellHighlightBorder"));
        else
            setBorder(null);
        return this;
    }
}
```

Planet	Radius	Image	Gaseous	Color	Image
Mars	3,397		<input type="checkbox"/>		
Jupiter	71,492		<input checked="" type="checkbox"/>		
Saturn	60,268		<input checked="" type="checkbox"/>		

Рис. 10.12. Таблица со средствами воспроизведения ячеек

Нетрудно заметить, что при получении ячейкой фокуса ввода средство воспроизведения устанавливает рамку. (Для установки нужной рамки запрашивается объект типа `UIManager`. А для выбора подходящего ключа поиска приходится обращаться к исходному коду класса `DefaultTableCellRenderer`.)

Как правило, для указания на то, что выбрана текущая ячейка таблицы, устанавливается походящий цвет фона. Здесь трудно посоветовать что-нибудь конкретное, поскольку не исключено, что фон будет плохо сочетаться с остальными цветами. Способ обозначения выбранной ячейки таблицы можно, в частности, позаимствовать из рассмотренного ранее примера программы `ListRenderingTest` из листинга 10.4.

Таблице необходимо каким-то образом указать, что для воспроизведения всех объектов типа `Color` следует использовать данное средство. Для этого предусмотрен метод `setDefaultRenderer()` из класса `JTable`. Ему передаются объект типа `Class` и требующееся средство воспроизведения следующим образом:

```
table.setDefaultRenderer(Color.class, new ColorTableCellRenderer());
```

В итоге указанное средство будет использоваться для воспроизведения всех объектов данного типа. Если средство воспроизведения требуется выбрать по какому-нибудь другому критерию, для этого придется создать подкласс, производный от класса `JTable`, а также переопределить метод `getCellRenderer()`.



Совет! Если конкретное средство воспроизведения должно просто выводить текстовую строку или пиктограмму, его можно построить в виде подкласса, расширяющего класс `DefaultTableCellRenderer`. В таком случае средства, реализованные в суперклассе, возьмут на себя ответственность за воспроизведение ячейки в состоянии выбора или обладания фокусом ввода.

10.2.4.2. Воспроизведение заголовков

Чтобы воспроизвести пиктограмму или иное изображение в заголовке столбца таблицы, необходимо установить соответствующее значение для этого заголовка следующим образом:

```
moonColumn.setHeaderValue(new ImageIcon("Moons.gif"));
```

Но ведь заголовок столбца таблицы не настолько логически развит, чтобы самостоятельно выбирать подходящее средство воспроизведения по указанному значению. Поэтому данное средство придется установить вручную. Например, для того чтобы показать пиктограмму в заголовке столбца, необходимо вызвать следующий метод:

```
moonColumn.setHeaderRenderer(table.getDefaultRenderer(  
    ImageIcon.class));
```

10.2.4.3. Редактирование ячеек таблицы

Чтобы разрешить редактирование ячеек таблицы, следует получить из модели таблицы сведения о том, какие именно ячейки можно редактировать. Для этой цели служит метод `isCellEditable()`. Чаще всего редактируемыми являются не отдельные ячейки, а целые столбцы. Например, в приведенном ниже фрагменте кода разрешается редактирование данных в четырех столбцах.

```
public boolean isCellEditable(int r, int c)  
{  
    return c == PLANET_COLUMN || c == MOONS_COLUMN ||  
        c == GASEOUS_COLUMN || c == COLOR_COLUMN;  
}
```

 **На заметку!** В классе `AbstractTableModel` имеется метод `isCellEditable()`, возвращающий логическое значение `false`, а в классе `DefaultTableModel` этот метод переопределен и по умолчанию возвращает логическое значение `true`.

В примере программы, исходный код которой представлен в листингах 10.8-10.11, можно устанавливать и сбрасывать флагки непосредственно в ячейках столбца Gaseous таблицы с данными о планетах. В этой программе можно также выбирать нужное количество спутников планет из комбинированных списков в ячейках столбца Moons, как показано на рис. 10.13. Ниже поясняется, как организовать редактор ячеек таблицы на основании комбинированного списка. И, наконец, если щелкнуть на любой ячейке в первом столбце Planet рассматриваемой здесь таблицы, то эта ячейка получит фокус ввода, а следовательно, в ней можно ввести или отредактировать название планеты.

В данном примере программы демонстрируется применение трех вариантов класса `DefaultCellEditor` для редактирования ячеек типа `JTextField`, `JCheckBox` и `JComboBox`. Класс `JTable` автоматически устанавливает редактор флагков для ячеек типа `Boolean`, а также редактор текстовых полей для всех редактируемых ячеек, у которых отсутствует собственное средство воспроизведения. В текстовых полях имеется возможность редактировать символьные строки, которые получаются в результате вызова метода `toString()` со значением, возвращаемым методом `getValueAt()` из модели таблицы.

Planet	Radius	Moons	Gaseous	Color	Image
Mercury	2,440	0	<input type="checkbox"/>		
Venus	5,052	0	<input type="checkbox"/>		
Earth	6,378	1	<input type="checkbox"/>		

Рис. 10.13. Редактирование ячеек таблицы

По завершении редактирования в результате вызова метода `getCellEditorValue()` отредактированное значение извлекается из соответствующего редактора. Этот метод должен возвратить значение правильного типа (т.е. того типа, который возвращается методом `getColumnType()` из модели).

Редактор ячейки таблицы на основании комбинированного списка придется создать вручную, потому что компоненту `JTable` неизвестно, какие именно значения могут подойти для отдельного вида ячейки. Так, в столбце `Moons` требуется организовать выбор любого значения в пределах от 0 до 20. Ниже приведен соответствующий код для инициализации этими значениями комбинированного списка.

```
JComboBox moonCombo = new JComboBox();
for (int i = 0; i <= 20; i++)
    moonCombo.addItem(i);
```

Чтобы создать объект класса `DefaultCellEditor` для такого типа данных, нужно передать комбинированный список конструктору этого класса следующим образом:

```
TableCellEditor moonEditor = new DefaultCellEditor(moonCombo);
```

Затем следует установить редактор, который, в отличие от средства воспроизведения цвета в столбце `Color`, не должен зависеть от *типа* объекта. Его совсем не обязательно использовать для всех объектов типа `Integer`, а достаточно установить только в отдельном столбце следующим образом:

```
moonColumn.setCellEditor(moonEditor);
```

10.2.4.4. Специальные редакторы

Запустите еще раз на выполнение рассматриваемую здесь программу и щелкните на ячейке столбца *Color*. На экране появится диалоговое окно селектора цвета планеты. Выберите нужный цвет и щелкните на кнопке *OK*. В итоге цвет ячейки изменится на выбранный (рис. 10.14).

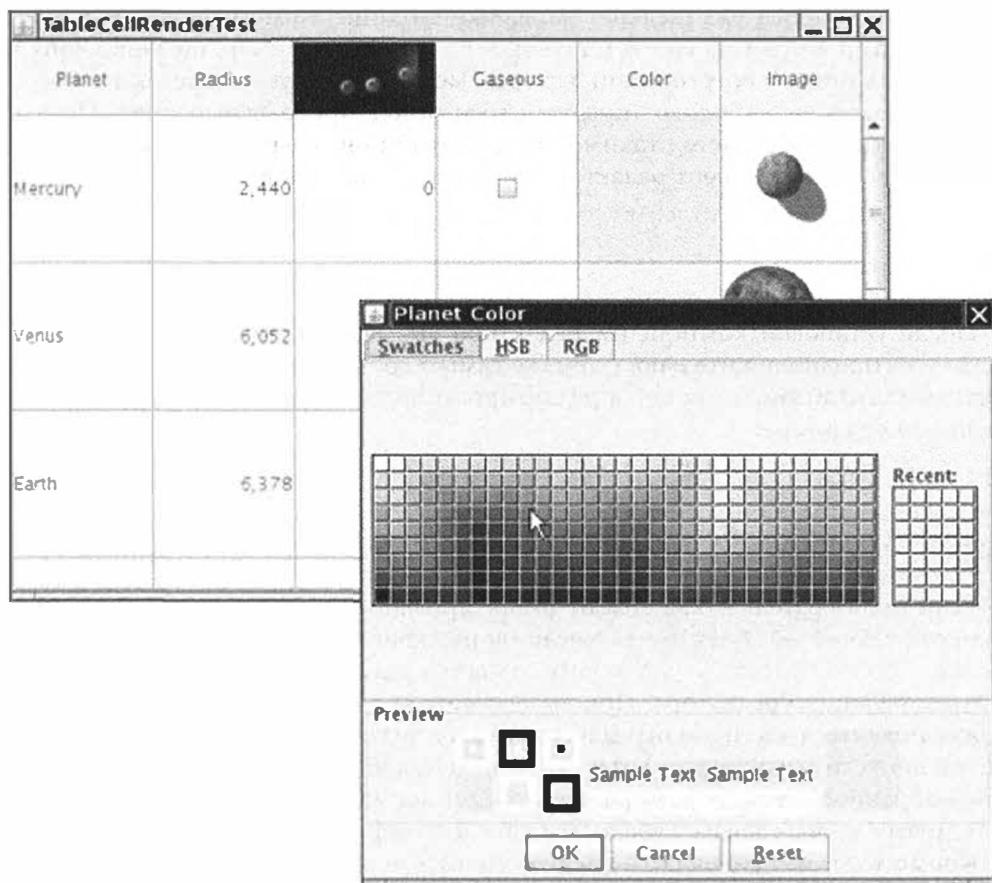


Рис. 10.14. Выбор цвета ячейки таблицы

Редактор цвета ячеек таблицы является не стандартным, а специальным, т.е. определяется и реализуется разработчиком прикладной программы. Для создания специального редактора ячеек таблицы следует реализовать интерфейс *TableCellEditor*. Впрочем, пользоваться этим интерфейсом не очень удобно, поэтому с версии JDK 1.3 для обработки событий предоставляется класс *AbstractCellEditor*.

Метод *getTableCellEditorComponent()* из интерфейса *TableCellEditor* запрашивает компонент разрешения на воспроизведение ячейки таблицы. Он действует таким же образом, как и метод *getTableCellRendererComponent()* из интерфейса *TableCellRenderer*, за исключением того, что у него отсутствует

параметр `focus`. Дело в том, что при редактировании ячейки таблицы предполагается, что эта ячейка обязательно имеет фокус ввода. На время редактирования компонент редактора временно заменяет средство воспроизведения. В данном случае возвращается пустая панель, для которой цвет не устанавливается. Это явно указывает пользователю на то, что в данный момент ячейка редактируется.

Далее требуется организовать отображение соответствующего редактора, как только пользователь щелкнет на ячейке таблицы. Компонент `JTable` вызывает редактор в ответ на соответствующее событие (например, щелчок кнопкой мыши). Для приема всех событий, которые могут инициировать редактирование, в классе `AbstractCellEditor` предусмотрен метод, приведенный ниже. Но если переопределить этот метод таким образом, чтобы он возвращал логическое значение `false`, то компонент редактора не будет установлен в таблице.

```
public boolean isCellEditable(EventObject anEvent)
{
    return true;
}
```

После установки компонента редактора вызывается метод `shouldSelectCell()` — предположительно, с тем же самым событием. В этом методе происходит инициализация процесса редактирования, например, отображается окно внешнего редактора:

```
public boolean shouldSelectCell(EventObject anEvent)
{
    colorDialog.setVisible(true);
    return true;
}
```

Если пользователь откажется от редактирования ячейки таблицы, вызывается метод `cancelCellEditing()`, а если он щелкнет на другой ячейке таблицы — метод `stopCellEditing()`. В обоих случаях следует закрыть диалоговое окно соответствующего редактора. При вызове метода `stopCellEditing()` в таблице может появиться частично отредактированное значение. Если такое значение является допустимым, следует возвратить логическое значение `true`. Любое значение, выбранное в окне селектора цвета, будет допустимым, но при редактировании других данных следует убедиться в их достоверности.

Кроме того, из суперкласса следует вызывать методы, отвечающие за инициализацию соответствующих событий. В противном случае редактирование не удастся отменить должным образом. В приведенном ниже фрагменте кода показано, как отменяется редактирование.

```
public void cancelCellEditing()
{
    colorDialog.setVisible(false);
    super.cancelCellEditing();
}
```

И наконец, необходимо определить метод, возвращающий значение, получающееся в результате редактирования:

```
public Object getCellEditorValue()
{
    return colorChooser.getColor();
}
```

Таким образом, к специальному редактору ячеек таблицы предъявляются следующие требования.

1. Он должен расширять класс `AbstractCellEditor` и реализовывать интерфейс `TableCellEditor`.
2. В нем должен быть определен метод `getTableCellEditorComponent()`, предназначенный для предоставления компонента редактора. Это может быть фиктивный компонент, если редактор предполагается отображать в отдельном диалоговом окне, или же компонент для редактирования непосредственно в ячейке, как, например, комбинированный список или текстовое поле.
3. В нем должны быть определены методы `shouldSelectCell()`, `stopCellEditing()` и `cancelCellEditing()` для управления запуском, завершением и отменой процесса редактирования. Для уведомления обработчиков событий из суперкласса должны быть вызваны методы `stopCellEditing()` и `cancelCellEditing()`.
4. В нем должен быть определен метод `getCellEditorValue()`, возвращающий значение, получаемое в результате редактирования.

Когда пользователь завершит редактирование, следует вызвать метод `stopCellEditing()` или `cancelCellEditing()`. Так, при создании диалогового окна селектора цвета устанавливаются приведенные ниже методы обратного вызова, уведомляющие о подтверждении или отмене результатов редактирования и инициирующие соответствующие события.

```
colorDialog = JColorChooser.createDialog(
    null, "Planet Color", false, colorChooser,
    EventHandler.create(ActionListener.class,
        this, "stopCellEditing"),
    EventHandler.create(ActionListener.class,
        this, "cancelCellEditing"));
```

На этом реализация специального редактора завершается. Теперь вы знаете, как сделать ячейку таблицы редактируемой и установить ее редактор. Остается открытым только один вопрос: как обновить модель с учетом отредактированного значения? По завершении редактирования компонент `JTable` вызывает следующий метод из модели таблицы:

```
void setValueAt(Object value, int r, int c)
```

Для сохранения нового значения этот метод придется переопределить. Параметр `value` теперь будет обозначать объект, возвращаемый редактором ячейки таблицы. В определении редактора ячейки таблицы задается тип объекта, который возвращается методом `getCellEditorValue()`. Так, при использовании класса `DefaultCellEditor` возможны три варианта указания типа для этого значения. Для редактора ячейки в виде флажка это тип `boolean`, для текстового поля — тип `String`, а для комбинированного списка — тип объекта, выбираемого пользователем.

Если объект `value` имеет другой тип, его необходимо привести к нужному типу. Чаще всего это происходит при редактировании чисел в текстовом поле.

В данном примере комбинированный список содержит объекты типа Integer, поэтому никакого приведения типов не требуется.

Листинг 10.8. Исходный код из файла `tableCellRender/TableCellRenderFrame.java`

```

1 package tableCellRender;
2
3 import java.awt.*;
4 import javax.swing.*;
5 import javax.swing.table.*;
6
7 /**
8  * Этот фрейм содержит таблицу с данными о планетах
9 */
10 public class TableCellRenderFrame extends JFrame
11 {
12     private static final int DEFAULT_WIDTH = 600;
13     private static final int DEFAULT_HEIGHT = 400;
14
15     public TableCellRenderFrame()
16     {
17         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
18
19         TableModel model = new PlanetTableModel();
20         JTable table = new JTable(model);
21         table.setRowSelectionAllowed(false);
22
23         // установить средства воспроизведения и
24         // редактирования ячеек таблицы
25         table.setDefaultRenderer(Color.class,
26             new ColorTableCellRenderer());
27         table.setDefaultEditor(Color.class,
28             new ColorTableCellEditor());
29         JComboBox<Integer> moonCombo = new JComboBox<>();
30         for (int i = 0; i <= 20; i++)
31             moonCombo.addItem(i);
32
33         TableColumnModel columnModel = table.getColumnModel();
34         TableColumn moonColumn =
35             columnModel.getColumn(PlanetTableModel.MOONS_COLUMN);
36         moonColumn.setCellEditor(new DefaultCellEditor(moonCombo));
37         moonColumn.setHeaderRenderer(
38             table.getDefaultRenderer(ImageIcon.class));
39         moonColumn.setHeaderValue(
40             new ImageIcon(getClass().getResource("Moons.gif")));
41
42         // показать таблицу
43         table.setRowHeight(100);
44         add(new JScrollPane(table), BorderLayout.CENTER);
45     }
46 }
```

Листинг 10.9. Исходный код из файла `tableCellRender/PlanetTableModel.java`

```

1 package tableCellRender;
2
3 import java.awt.*;
```

```
4 import javax.swing.*;
5 import javax.swing.table.*;
6
7 /**
8  * Модель таблицы планет, определяющая значения, свойства
9  * воспроизведения и редактирования данных о планетах
10 */
11 public class PlanetTableModel extends AbstractTableModel
12 {
13     public static final int PLANET_COLUMN = 0;
14     public static final int MOONS_COLUMN = 2;
15     public static final int GASEOUS_COLUMN = 3;
16     public static final int COLOR_COLUMN = 4;
17
18     private Object[][] cells = {
19         { "Mercury", 2440.0, 0, false, Color.YELLOW,
20             new ImageIcon(getClass().getResource("Mercury.gif")) },
21         { "Venus", 6052.0, 0, false, Color.YELLOW,
22             new ImageIcon(getClass().getResource("Venus.gif")) },
23         { "Earth", 6378.0, 1, false, Color.BLUE,
24             new ImageIcon(getClass().getResource("Earth.gif")) },
25         { "Mars", 3397.0, 2, false, Color.RED,
26             new ImageIcon(getClass().getResource("Mars.gif")) },
27         { "Jupiter", 71492.0, 16, true, Color.ORANGE,
28             new ImageIcon(getClass().getResource("Jupiter.gif")) },
29         { "Saturn", 60268.0, 18, true, Color.ORANGE,
30             new ImageIcon(getClass().getResource("Saturn.gif")) },
31         { "Uranus", 25559.0, 17, true, Color.BLUE,
32             new ImageIcon(getClass().getResource("Uranus.gif")) },
33         { "Neptune", 24766.0, 8, true, Color.BLUE,
34             new ImageIcon(getClass().getResource("Neptune.gif")) },
35         { "Pluto", 1137.0, 1, false, Color.BLACK,
36             new ImageIcon(getClass().getResource("Pluto.gif")) } };
37
38     private String[] columnNames =
39         { "Planet", "Radius", "Moons", "Gaseous", "Color", "Image" };
40
41     public String getColumnName(int c)
42     {
43         return columnNames[c];
44     }
45
46     public Class<?> getColumnClass(int c)
47     {
48         return cells[0][c].getClass();
49     }
50
51     public int getColumnCount()
52     {
53         return cells[0].length;
54     }
55
56     public int getRowCount()
57     {
58         return cells.length;
59     }
60
61     public Object getValueAt(int r, int c)
```

```
62     {
63         return cells[r][c];
64     }
65
66     public void setValueAt(Object obj, int r, int c)
67     {
68         cells[r][c] = obj;
69     }
70     public boolean isCellEditable(int r, int c)
71     {
72         return c == PLANET_COLUMN || c == MOONS_COLUMN ||
73                c == GASEOUS_COLUMN || c == COLOR_COLUMN;
74     }
75 }
```

Листинг 10.10. Исходный код из файла `tableCellRender/ColorTableCellRenderer.java`

```
1 package tableCellRender;
2
3 import java.awt.*;
4 import javax.swing.*;
5 import javax.swing.table.*;
6
7 /**
8  * Это средство воспроизведения отображает цвет
9  * в виде панели с заданным цветом
10 */
11 public class ColorTableCellRenderer extends
12     JPanel implements TableCellRenderer
13 {
14     public Component getTableCellRendererComponent(
15             JTable table, Object value, boolean isSelected,
16             boolean hasFocus, int row, int column)
17     {
18         setBackground((Color) value);
19         if (hasFocus) setBorder(
20             UIManager.getBorder("Table.focusCellHighlightBorder"));
21         else setBorder(null);
22         return this;
23     }
24 }
```

Листинг 10.11. Исходный код из файла `tableCellRender/ColorTableCellEditor.java`

```
1 package tableCellRender;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.beans.*;
6 import java.util.*;
7 import javax.swing.*;
8 import javax.swing.table.*;
9
```

```
10 /**
11  * Этот редактор открывает диалоговое окно селектора цвета
12  * для редактирования значения цвета в выбранной ячейке таблицы
13 */
14 public class ColorTableCellEditor extends
15     AbstractCellEditor implements TableCellEditor
16 {
17     private JColorChooser colorChooser;
18     private JDialog colorDialog;
19     private JPanel panel;
20     public ColorTableCellEditor()
21     {
22         panel = new JPanel();
23         // подготовить диалоговое окно выбора цвета
24
25         colorChooser = new JColorChooser();
26         colorDialog = JColorChooser.createDialog(
27             null, "Planet Color", false, colorChooser,
28             EventHandler.create(
29                 ActionListener.class, this, "stopCellEditing"),
30             EventHandler.create(
31                 ActionListener.class, this, "cancelCellEditing"));
32     }
33
34
35     public Component getTableCellEditorComponent(
36         JTable table, Object value, boolean isSelected,
37         int row, int column)
38     {
39         // Именно здесь получается текущее значение цвета,
40         // сохраняемое в диалоговом окне на тот случай,
41         // если пользователь начнет редактирование
42         colorChooser.setColor((Color) value);
43         return panel;
44     }
45
46     public boolean shouldSelectCell(EventObject anEvent)
47     {
48         // начать редактирование
49         colorDialog.setVisible(true);
50
51         // уведомить вызывающую часть программы о том, что
52         // эту ячейку разрешается выбрать
53         return true;
54     }
55
56     public void cancelCellEditing()
57     {
58         // редактирование отменено - скрыть диалоговое окно
59         colorDialog.setVisible(false);
60         super.cancelCellEditing();
61     }
62
63     public boolean stopCellEditing()
64     {
65         // редактирование завершено - скрыть диалоговое окно
66         colorDialog.setVisible(false);
67         super.stopCellEditing();
68     }
```

```

69     // уведомить вызывающую часть программы о том, что
70     // данное значение цвета разрешается использовать
71     return true;
72 }
73 public Object getCellEditorValue()
74 {
75     return colorChooser.getColor();
76 }
77 }
```

javax.swing.JTable 1.2

- TableCellRenderer getDefaultRenderer(Class<?> type)**
Получает средство воспроизведения, выбираемое по умолчанию для указанного типа ячейки таблицы.
- TableCellEditor getDefaultEditor(Class<?> type)**
Получает редактор, выбираемый по умолчанию для указанного типа ячейки таблицы.

javax.swing.table.TableCellRenderer 1.2

- Component getTableCellRendererComponent(JTable table, Object value, boolean selected, boolean hasFocus, int row, int column)**
Возвращает компонент, метод **paint()** которого вызывается для воспроизведения ячейки таблицы.

Параметры:

table	Таблица, содержащая воспроизведимые ячейки
value	Воспроизведимая ячейка
selected	Принимает логическое значение true , если выбрана текущая ячейка
hasFocus	Принимает логическое значение true , если текущая ячейка обладает фокусом ввода
row, column	Номер строки и столбца с воспроизведимой ячейкой

javax.swing.table.TableColumn 1.2

- void setCellEditor(TableCellEditor editor)**
- void setCellRenderer(TableCellRenderer renderer)**
Устанавливают редактор и средство воспроизведения для всех ячеек в указанном столбце.
- void setHeaderRenderer(TableCellRenderer renderer)**
Устанавливает средство воспроизведения для ячейки с заголовком в указанном столбце.
- void setHeaderValue(Object value)**
Устанавливает значение, которое должно быть отображено в виде заголовка в данном столбце.

javax.swing.DefaultCellEditor 1.2

- **DefaultCellEditor(JComboBox comboBox)**

Создает редактор, предоставляющий комбинированный список для выбора значений ячеек таблицы.

javax.swing.table.TableCellEditor 1.2

- **Component getTableCellEditorComponent(JTable table, Object value, boolean selected, int row, int column)**

Возвращает компонент, метод **paint()** которого вызывается для воспроизведения ячейки таблицы.

Параметры:

table	Таблица, содержащая воспроизводимые ячейки
value	Воспроизводимая ячейка
selected	Принимает логическое значение
row, column	true , если текущая ячейка выбрана Номер строки и столбца с воспроизводимой ячейкой

javax.swing.CellEditor 1.2

- **boolean isCellEditable(EventObject event)**

Возвращает логическое значение **true**, если событие пригодно для запуска процесса редактирования данной ячейки.

- **boolean shouldSelectCell(EventObject anEvent)**

Запускает процесс редактирования. Как правило, возвращается логическое значение **true**, если должна быть выбрана редактируемая ячейка. Если же требуется, чтобы содержимое выбранной ячейки не изменилось в результате редактирования, следует возвратить логическое значение **false**.

- **void cancelCellEditing()**

Отменяет процесс редактирования. Его результаты можно проигнорировать.

- **boolean stopCellEditing()**

Останавливает процесс редактирования с намерением использовать его результаты. Возвращает логическое значение **true**, если значение, получаемое в результате редактирования, оказывается допустимым.

- **Object getCellEditorValue()**

Возвращает результаты редактирования.

- **void addCellEditorListener(CellEditorListener l)**

- **void removeCellEditorListener(CellEditorListener l)**

Вводят или удаляют обязательный приемник событий, наступающих при редактировании ячеек таблицы.

10.3. Деревья

Каждому пользователю компьютера, оперирующего иерархической файловой системой, знакомы древовидные представления файлов и каталогов. Это, конечно, лишь один из многих примеров применения древовидной структуры. В повседневной жизни такую же структуру образует система административно-территориального деления страны на штаты, области и города (рис. 10.15).

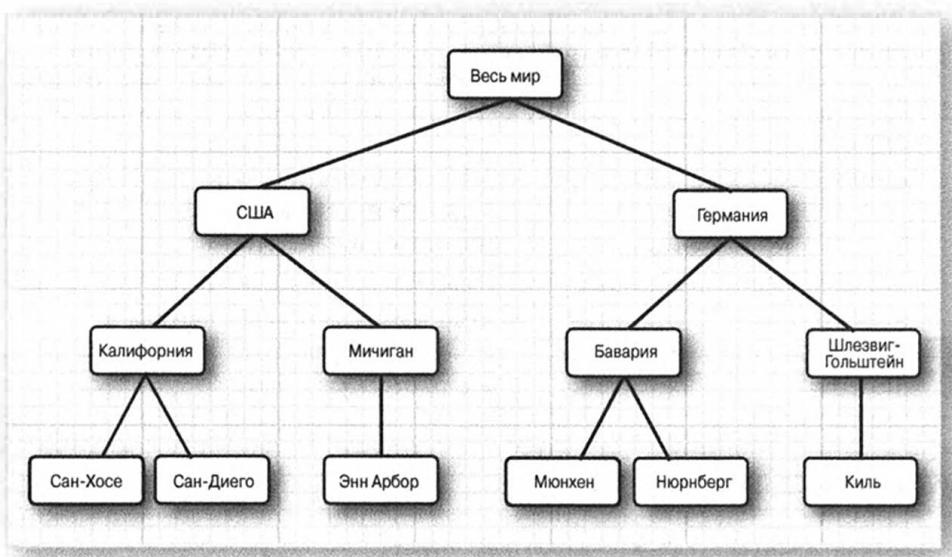


Рис. 10.15. Древовидная система административно-территориального деления страны на штаты, области и города

Программирующим на Java нередко приходится писать код для отображения подобных структур. И для этого в библиотеке Swing предусмотрен класс `JTree`. Вместе со вспомогательными классами он берет на себя все хлопоты по компоновке древовидной структуры и обработке запросов пользователей на развертывание и свертывание узлов дерева. В этом разделе показано, как пользоваться средствами, доступными в классе `JTree`, для построения древовидных структур.

Подобно другим сложным компонентам Swing, здесь рассматриваются только самые общие и наиболее распространенные приемы обращения с деревьями. А для углубленного изучения данного вопроса рекомендуется упоминавшаяся ранее дополнительная литература: *Graphic Java™: Mastering the JFC, Volume II: Swing, 3rd Edition* Дэвида М. Гери или *Core Swing* Кима Топли.

Приступая к рассмотрению деревьев, необходимо сначала пояснить терминологию, характерную для них (рис. 10.16). Прежде всего, дерево состоит из узлов. Каждый узел может быть листом (т.е. краевым узлом) или иметь дочерние узлы. У каждого узла, кроме корневого, имеется только один родительский узел. А у дерева в целом имеется только один корневой узел. Ряд деревьев с собственными корневыми узлами называется лесом.

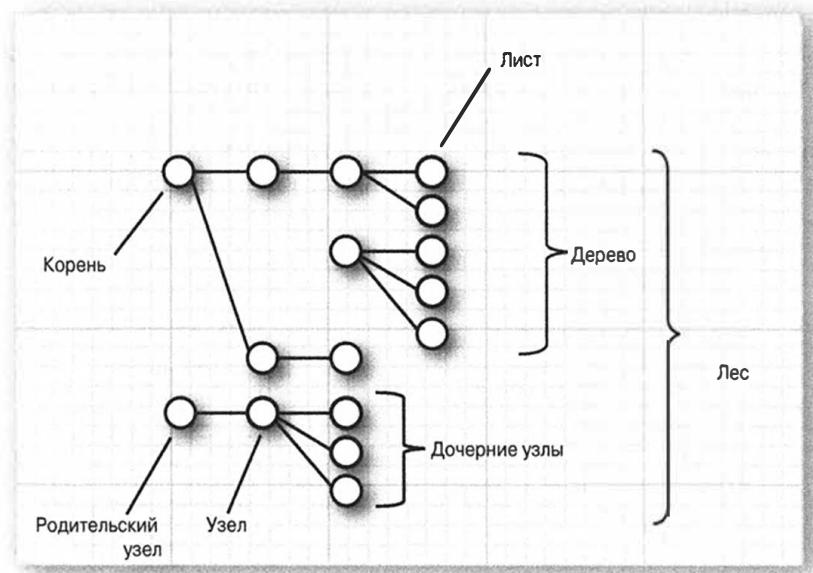


Рис. 10.16. Терминология, употребляемая для описания древовидных структур

10.3.1. Простые деревья

В первом несложном примере программы будет построено дерево лишь с несколькими узлами, как показано далее на рис. 10.18. Подобно многим другим компонентам библиотеки Swing, программирующий на Java создает модель иерархических данных, а компонент JTree автоматически отображает их. Для создания объекта типа JTree его конструктору передается модель дерева следующим образом:

```
TreeModel model = . . . ;
JTree tree = new JTree(model);
```



На заметку! Некоторые конструкторы создают деревья из коллекции составляющих элементов:

```
JTree(Object[] nodes)
JTree(Vector<?> nodes)
JTree(Hashtable<?, ?> nodes) // значения становятся узлами
```

Такие конструкторы практически бесполезны, потому что они создают только лес деревьев, каждое из которых содержит единственный узел. А третий конструктор из тех, что приведены выше, и вовсе не приносит никакой пользы, поскольку узлы передаются ему в произвольном порядке, который определяется хеш-кодами ключей.

Как же получить модель дерева? С помощью класса, реализующего интерфейс TreeModel, можно создать собственную модель. Такой способ будет рассмотрен далее в этой главе, а до тех пор воспользуемся моделью дерева типа DefaultTreeModel, предоставляемой в библиотеке Swing по умолчанию. Для построения используемой по умолчанию модели дерева необходимо предоставить конструктору корневой узел следующим образом:

```
TreeNode root = . . .;
DefaultTreeModel model = new DefaultTreeModel(root);
```

где `TreeNode` — это еще один интерфейс. Используемую по умолчанию модель можно заполнить экземплярами любого класса, реализующего данный интерфейс. В данном случае применяется уже готовый класс узлов `DefaultMutableTreeNode` из библиотеки Swing. Как показано на рис. 10.17, этот класс реализует интерфейс `MutableTreeNode`, который является дочерним по отношению к интерфейсу `TreeNode`.

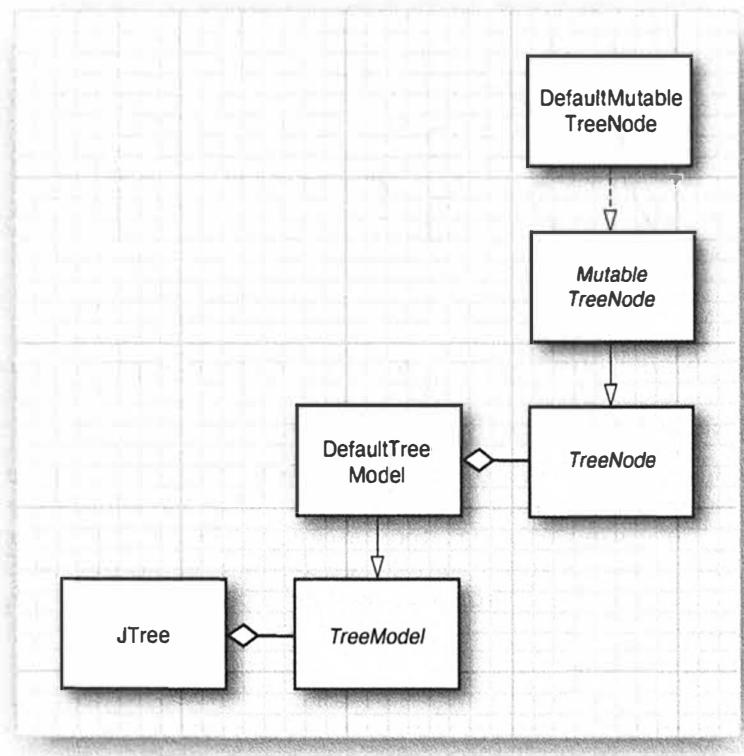


Рис. 10.17. Иерархия наследования интерфейсов и классов, используемых при построении деревьев

Изменяемый по умолчанию узел дерева (т.е. экземпляр класса `DefaultMutableTreeNode`) содержит пользовательский объект. Эти объекты воспроизводятся для всех узлов. Если не задано специальное средство воспроизведения, то в дереве отображается символьная строка, получаемая в результате выполнения метода `toString()`.

В первом рассматриваемом здесь примере построения деревьев в качестве пользовательских объектов применяются символьные строки, хотя на практике древовидные структуры обычно заполняются более сложными объектами. Например, для отображения дерева каталогов в качестве его узлов имеет смысл использовать объекты типа `File`. Пользовательский объект можно указать сразу

в конструкторе или в дальнейшем с помощью метода `setUserObject()`, как показано ниже.

```
DefaultMutableTreeNode node = new DefaultMutableTreeNode("Texas");
...
node.setUserObject("California");
```

Затем между узлами следует установить отношения “родительский–дочерний”. Для этого нужно ввести дочерние узлы с помощью метода `add()`, начиная с корневого узла, как показано в приведенном ниже фрагменте кода. Получаемое в итоге простое дерево приведено на рис. 10.18.

```
DefaultMutableTreeNode root = new DefaultMutableTreeNode("World");
DefaultMutableTreeNode country = new DefaultMutableTreeNode("USA");
root.add(country);
DefaultMutableTreeNode state =
    new DefaultMutableTreeNode("California");
country.add(state);
country.add(new DefaultMutableTreeNode("San Jose"));
country.add(new DefaultMutableTreeNode("Cupertino"));
DefaultMutableTreeNode city =
    new DefaultMutableTreeNode("Michigan");
city.add(new DefaultMutableTreeNode("Ann Arbor"));
root.add(city);
root.add(new DefaultMutableTreeNode("Germany"));
root.add(new DefaultMutableTreeNode("Schleswig-Holstein"));
```



Рис. 10.18. Простое дерево

Подобным образом следует связать все узлы, а затем построить модель дерева типа `DefaultTreeModel` с корневым узлом, а на основе этой модели — само дерево с помощью компонента `JTree`:

```
DefaultTreeModel treeModel = new DefaultTreeModel(root);
JTree tree = new JTree(treeModel);
```

Можно поступить еще проще, передав корневой узел конструктору класса `JTree`, который автоматически создаст модель дерева по умолчанию, как показано ниже. Весь исходный код рассматриваемого здесь примера программы, демонстрирующей построение простого дерева, приведен в листинге 10.12.

```
JTree tree = new JTree(root);
```

Листинг 10.12. Исходный код из файла `tree/SimpleTreeFrame.java`

```
1 package tree;
2
3 import javax.swing.*;
```

```
4 import javax.swing.tree.*;
5
6 /**
7  * Этот фрейм содержит простое дерево, отображающее
8  * построенную вручную модель дерева
9 */
10 public class SimpleTreeFrame extends JFrame
11 {
12     private static final int DEFAULT_WIDTH = 300;
13     private static final int DEFAULT_HEIGHT = 200;
14
15     public SimpleTreeFrame()
16     {
17         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
18
19         // подготовить данные для модели дерева
20
21         DefaultMutableTreeNode root =
22             new DefaultMutableTreeNode("World");
23         DefaultMutableTreeNode country =
24             new DefaultMutableTreeNode("USA");
25         root.add(country);
26         DefaultMutableTreeNode state =
27             new DefaultMutableTreeNode("California");
28         country.add(state);
29         DefaultMutableTreeNode city =
30             new DefaultMutableTreeNode("San Jose");
31         state.add(city);
32         city = new DefaultMutableTreeNode("Cupertino");
33         state.add(city);
34         state = new DefaultMutableTreeNode("Michigan");
35         country.add(state);
36         city = new DefaultMutableTreeNode("Ann Arbor");
37         state.add(city);
38         country = new DefaultMutableTreeNode("Germany");
39         root.add(country);
40         state = new DefaultMutableTreeNode("Schleswig-Holstein");
41         country.add(state);
42         city = new DefaultMutableTreeNode("Kiel");
43         state.add(city);
44
45         // построить дерево и разместить его
46         // на прокручиваемой панели
47         JTree tree = new JTree(root);
48         add(new JScrollPane(tree));
49     }
50 }
```

Дерево, отображаемое при выполнении данной программы, показано на рис. 10.19. В рабочем окне программы будут видны только корневой узел и его дочерние узлы. Для развертывания поддеревьев следует щелкнуть мышью на пиктограмме кружка (*маркер* узла). Линия, соединяющаяся с кружком, направлена вправо, если поддерево свернуто, или вниз, если оно развернуто (рис. 10.20). Неизвестно, что имели в виду разработчики визуального стиля Metal, но, по-видимому, кружок, соединяемый с линией, обозначает дверную ручку, которую нужно повернуть вниз по часовой стрелке, чтобы “открыть” поддерево.

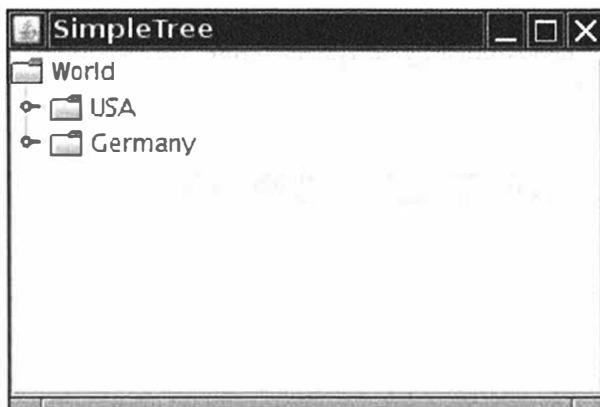


Рис. 10.19. Исходное состояние отображаемого дерева

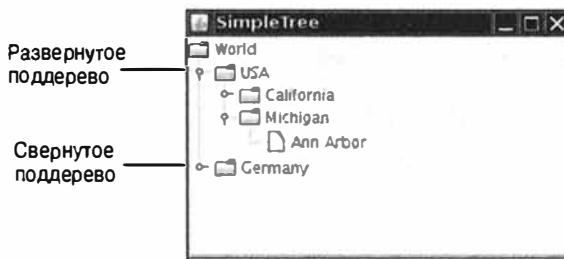


Рис. 10.20. Свернутые и развернутые поддеревья

На заметку! Разумеется, внешний вид дерева зависит от выбранного визуального стиля. Приведенное выше описание относится к визуальному стилю Metal. А в визуальных стилях Windows и Motif для обозначения свернутого и развернутого дерева используются знаки + и - (рис. 10.21).

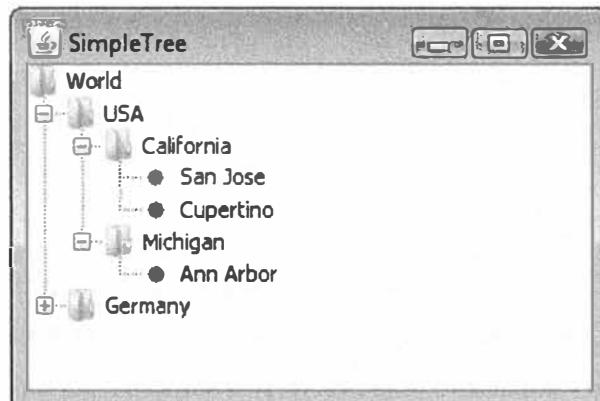


Рис. 10.21. Вид дерева, отображаемого в визуальном стиле Windows

Чтобы скрыть линии, связывающие родительские и дочерние узлы, как показано на рис. 10.22, необходимо указать значение `None` свойства `JTree.lineStyle` следующим образом:

```
tree.putClientProperty("JTree.lineStyle", "None");
```

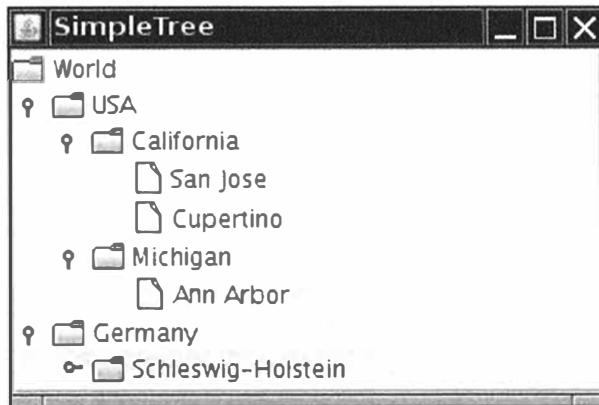


Рис. 10.22. Вид дерева без соединительных линий между узлами

А для отображения этих линий нужно указать значение `Angled` данного свойства:

```
tree.putClientProperty("JTree.lineStyle", "Angled");
```

На рис. 10.23 показан еще один стиль представления структуры дерева с помощью горизонтальных линий. Такое дерево отображается с горизонтальными линиями, разделяющими только дочерние узлы корневого узла. Но трудно себе представить ситуацию, когда могла бы пригодиться такая древовидная структура.



Рис. 10.23. Вид дерева с разделяющими горизонтальными линиями

По умолчанию у корневого узла отсутствует маркер для сворачивания всего дерева. Если требуется этот маркер, то следует вызвать приведенный ниже метод.

На рис. 10.24 представлен результат вызова данного метода. Теперь все дерево можно сворачивать и разворачивать полностью.

```
tree.setShowsRootHandles(true);
```

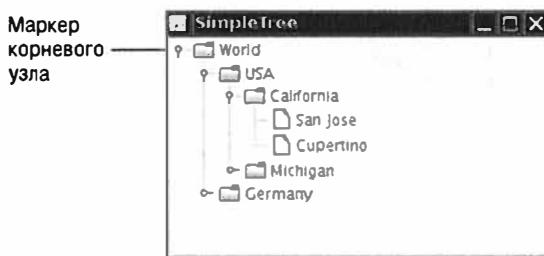


Рис. 10.24. Дерево с маркером корневого узла

Кроме того, корневой узел можно скрыть. Это может пригодиться, например, для отображения леса, т.е. ряда деревьев с собственными корневыми узлами. Все деревья следует объединить в дереве с общим корнем, а затем скрыть этот общий корень с помощью следующего метода:

```
tree.setRootVisible(false);
```

На рис. 10.25 приведен пример леса с двумя деревьями, имеющими корневые узлы USA и Germany, объединенные в одном дереве со скрытым корневым узлом. Перейдем теперь от корня к листьям дерева. Для их отображения используется пиктограмма листа бумаги, как показано на рис. 10.26.

Итак, каждый узел отображается отдельной пиктограммой. Для обозначения узлов дерева имеются три вида пиктограмм: листа бумаги, открытой и закрытой папки. Средству воспроизведения узлов должно быть известно, какой именно пиктограммой следует отображать каждый узел. По умолчанию решение принимается следующим образом: если метод `isLeaf()` возвращает логическое значение `true`, то используется пиктограмма листа бумаги, в противном случае — пиктограмма папки.



Рис. 10.25. Вид леса

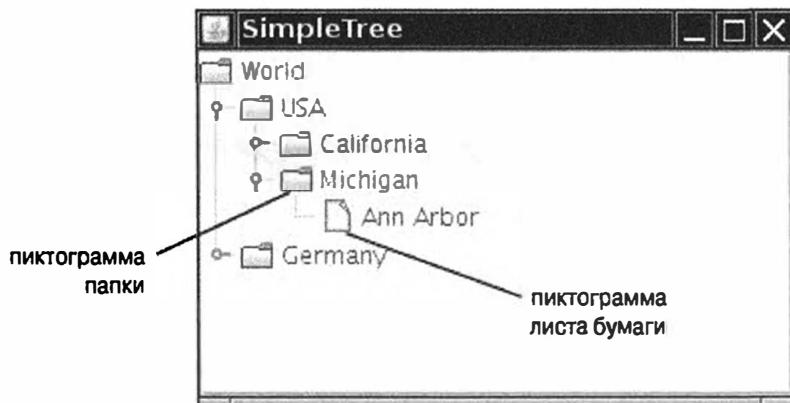


Рис. 10.26. Пиктограммы папок и листов бумаги

Метод `isLeaf()` из класса `DefaultMutableTreeNode` возвращает логическое значение `true`, если у данного узла отсутствуют дочерние узлы. Таким образом, узлы дерева с дочерними узлами будут отображаться в виде папок, а узлы дерева без дочерних узлов — в виде листов бумаги.

Но такой способ обозначения узлов дерева подходит не для всех случаев. Например, при добавлении узла `Montana` в дерево для отображения штата Монтана без указания городов этот штат будет обозначен пиктограммой листа бумаги. Но при этом будет нарушен сам принцип представления дерева, по которому такие пиктограммы служат для обозначения городов.

Компоненту `JTree` неизвестно, какие именно узлы являются листами дерева, поэтому для выяснения этого факта он обращается к модели дерева. Если же узел без дочерних узлов не является листом дерева в принципе, то для определения листов можно выбрать другой критерий, например, обратиться к свойству узла, определяющему допустимость в нем дочерних узлов. Так, если для некоторого узла дочерние узлы недопустимы, то нужно вызвать следующий метод:

```
node.setAllowsChildren(false);
```

Затем необходимо запросить модель дерева, чтобы она выяснила с помощью свойства допустимости дочерних узлов, является ли узел листом дерева и следует ли отображать его пиктограммой листа бумаги. С этой целью необходимо вызвать метод `setAsksAllowsChildren()` из класса `DefaultTreeModel` следующим образом:

```
model.setAsksAllowsChildren(true);
```

В таком случае те узлы, где допускается наличие дочерних узлов, будут обозначены пиктограммами папок, а те узлы, где не допускается наличие дочерних узлов, — пиктограммами листов бумаги. С другой стороны, задавая в конструкторе дерева корневой узел, можно также указать на необходимость запрашивать свойство допустимости дочерних узлов:

```
JTree tree = new JTree(root, true); // те узлы, где не допускаются
// дочерние узлы, обозначаются пиктограммами листов бумаги
```

javax.swing.JTree 1.2

- **JTree (TreeModel model)**
Конструирует дерево из указанной модели.
- **JTree (TreeNode root)**
- **JTree (TreeNode root, boolean asksAllowChildren)**
Конструируют дерево с заданной по умолчанию моделью, отображающее корневой узел и его дочерние узлы.

Параметры: root asksAllowChildren	Корневой узел Логическое значение true предписывает использовать свойство допустимости дочерних узлов, чтобы выяснить, является ли узел листом дерева
--	--
- **void setShowsRootHandles (boolean b)**
Если параметр **b** принимает логическое значение **true**, то в корневом узле дерева отображается маркер свертывания.
- **void setRootVisible (boolean b)**
Если параметр **b** принимает логическое значение **true**, то корневой узел отображается, а иначе он скрывается.

javax.swing.tree.TreeNode 1.2

- **boolean isLeaf()**
Возвращает логическое значение **true**, если данный узел является листом.
- **boolean getAllowsChildren()**
Возвращает логическое значение **true**, если данный узел может иметь дочерние узлы.

javax.swing.tree.MutableTreeNode 1.2

- **void setUserObject (Object userObject)**
Задает пользовательский объект типа **userObject**, используемый для воспроизведения узла дерева.

javax.swing.tree.TreeModel 1.2

- **boolean isLeaf (Object node)**
Возвращает логическое значение **true**, если узел **node** следует отобразить как лист дерева.

javax.swing.tree.DefaultTreeModel 1.2

- **void setAsksAllowsChildren(boolean b)**

Если параметр **b** принимает логическое значение **true**, то узлы отображаются как листья дерева, при условии, что метод **getAllowsChildren()** возвращает логическое значение **false**. В противном случае такой внешний вид узлов будет выбран, если метод **isLeaf()** возвратит логическое значение **true**.

javax.swing.tree.DefaultMutableTreeNode 1.2

- **DefaultMutableTreeNode(Object userObject)**

Создает изменяемый узел дерева с указанным пользовательским объектом.

- **void add(MutableTreeNode child)**

Вводит узел как последний дочерний узел в данном узле дерева.

- **void setAllowsChildren(boolean b)**

Если параметр **b** принимает логическое значение **true**, в данный узел могут быть введены дочерние узлы.

javax.swing.JComponent 1.2

- **void putClientProperty(Object key, Object value)**

Вводит указанную пару "ключ-значение" в небольшую таблицу, которой управляет каждый компонент. Этот "запасной" механизм используется в некоторых компонентах библиотеки Swing для хранения специальных свойств, определяющих их внешний вид.

10.3.2. Редактирование деревьев и путей к деревьям

В следующем примере программы демонстрируются способы редактирования деревьев. На рис. 10.27 показан пользовательский интерфейс данной программы, где для создания нового узла с именем New предусмотрены кнопки Add Sibling (Добавить равноправный узел) и Add Child (Добавить дочерний узел). Для удаления текущего выбранного узла предназначена кнопка Delete (Удалить).

Для реализации такого поведения следует найти текущий выбранный узел. В компоненте JTree предусмотрен интересный способ поиска такого узла по *пути к объекту*, называемого иначе *путем к дереву*. Такой путь начинается с корневого узла и состоит из последовательности дочерних узлов (характерный его пример приведен на рис. 10.28).

Казалось бы, для поиска текущего узла лучше было бы воспользоваться интерфейсом TreeNode и методом getParent(). Но на самом деле компоненту JTree ничего неизвестно об интерфейсе TreeNode, поскольку он используется только для реализации в классе DefaultTreeModel, а не в интерфейсе TreeModel. Дерево может иметь модель, в узлах которой не реализуется интерфейс TreeNode. Например, в модели дерева с другими типами объектов могут вообще отсутствовать

методы `getParent()` и `getChild()`. В таком случае для организации связей между узлами применяется модель дерева, но компоненту `JTree` ничего неизвестно о них. Поэтому в компоненте `JTree` предполагается всегда использовать полные пути к деревьям.



Рис. 10.27. Рабочее окно программы, демонстрирующей способы редактирования дерева

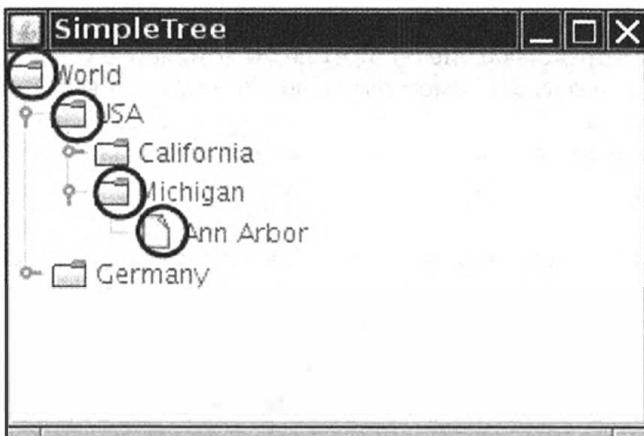


Рис. 10.28. Пример пути к дереву

Класс `TreePath` управляет последовательностью ссылок на объекты типа `Object` (а не `TreeNode`) с помощью нескольких методов. Например, для получения ссылки на последний узел пути можно вызвать метод `getLastPathComponent()`. А для поиска текущего выбранного узла служит метод `getSelectionPath()` из класса `JTree`. Таким образом, зная путь к дереву (на основании объекта типа `TreePath`), можно получить ссылку на текущий выбранный узел следующим образом:

```
TreePath selectionPath = tree.getSelectionPath();
DefaultMutableTreeNode selectedNode =
    (DefaultMutableTreeNode) selectionPath.getLastPathComponent();
```

Поскольку такой запрос выполняется очень часто, для него разработан служебный метод

```
DefaultMutableTreeNode selectedNode =
    (DefaultMutableTreeNode) tree.getLastSelectedPathComponent();
```

Этот метод называется `getLastSelectedPathComponent()`, а не `getSelectedNode()`, так как самому дереву ничего неизвестно об узлах, а его модель оперирует только путями к объектам.



На заметку! Кроме путей к деревьям, для описания узлов дерева используются методы из класса `JTree`, которые принимают или возвращают целочисленный индекс, обозначающий позицию строки (т.е. номер строки, начиная с нуля) для узла во внутреннем представлении дерева. Такие номера имеют только видимые узлы, причем номер строки может меняться при свертывании, развертывании или изменении дерева. Поэтому номера, обозначающие позиции строк, не рекомендуется применять для доступа к узлам. У каждого метода из класса `JTree`, предназначенного для работы с позициями строк, имеется эквивалентный метод, оперирующий путями к деревьям.

Получив выбранный узел, можно приступать к его редактированию. Но добавить к нему дочерние узлы нельзя, просто вызвав метод `add()`, как показано ниже.

```
selectedNode.add(newNode); // Нельзя!
```

Если изменяется структура узлов, то изменения вносятся в модель дерева, а связанное с ним представление об этом не уведомляется. Следовательно, такое уведомление нужно послать самостоятельно. Но если для ввода нового узла вызвать метод `insertNodeInto()` из класса `DefaultTreeModel`, то такое уведомление будет отправлено представлению дерева автоматически:

```
model.insertNodeInto(newNode, selectedNode,
    selectedNode.getChildCount());
```

Аналогичным образом применяется метод `removeNodeFromParent()` для удаления узла и уведомления об обновлении представления дерева:

```
model.removeNodeFromParent(selectedNode);
```

Для изменения пользовательского объекта, но с сохранением структуры узла необходимо вызвать метод `nodeChanged()` следующим образом:

```
model.nodeChanged(changedNode);
```

Автоматическое уведомление является основным достоинством модели типа `DefaultTreeModel`. При создании собственных моделей деревьев такое уведомление приходится организовывать самостоятельно. Более подробно этот вопрос обсуждается в упоминавшей ранее книге *Core Swing* Кима Топли.



Внимание! В состав класса `DefaultTreeModel` входит метод `reload()`, полностью перезагружающий модель дерева, но его не следует вызывать только для обновления дерева после внесения нескольких изменений в нее. Дело в том, что при таком обновлении дерева все узлы за пределами дочерних узлов корневого узла снова будут свернуты. Такое поведение дерева может оказаться неудобным для пользователей, вынуждая их всякий раз развертывать дерево.

Если представление получает уведомление об изменении структуры узлов дерева, оно обновляет отображение узлов, не развертывая их для просмотра вновь добавленных узлов. Например, добавление в рассматриваемом здесь примере программы нового дочернего узла в свернутые узлы произойдет незаметно для пользователя. В таком случае придется специально организовать развертывание родительских узлов для отображения введенного нового дочернего узла. Для этого из класса `JTree` можно вызвать метод `makeVisible()`, принимающий путь, ведущий к отображаемому на экране узлу.

Таким образом, для отображения вновь введенного узла придется сформировать путь к нему от корневого узла дерева. Для получения этого пути сначала следует вызвать метод `getPathToRoot()` из класса `DefaultTreeModel`. Он возвращает массив `TreeArray[]` для всех узлов (от текущего до корневого), который далее передается конструктору класса `TreePath`. В приведенном ниже фрагменте кода показано, каким образом вновь введенный узел становится видимым.

```
TreeNode[] nodes = model.getPathToRoot(newNode);
TreePath path = new TreePath(nodes);
tree.makeVisible(path);
```



На заметку! Любопытно, что класс `DefaultTreeModel` ведет себя так, как будто ему вообще ничего неизвестно о классе `TreePath`, хотя он и предназначен для взаимодействия с классом `JTree`. В то же время в классе `JTree` широко применяются пути и вообще не используются массивы узлов.

А теперь допустим, что дерево находится на прокручиваемой панели. После развертывания дерева новый узел все еще может быть за пределами текущего окна просмотра. Для перехода к новому узлу вместо метода `makeVisible()` следует вызвать метод `scrollPathToVisible()`, как показано ниже. Этот метод развертывает все узлы, указанные в заданном пути, прокручивая содержимое окна вплоть до последнего узла в конце пути (рис. 10.29).

```
tree.scrollPathToVisible(path);
```

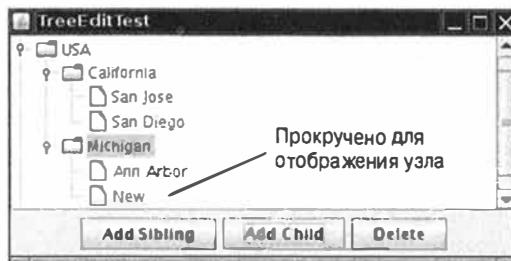


Рис. 10.29. Автоматическая прокрутка дерева на панели для просмотра нового узла

После двойного щелчка кнопкой мыши откроется окно редактора ячеек, вызываемого по умолчанию (рис. 10.30). Для его реализации служит класс `DefaultCellEditor`. А для редактирования узлов с иными объектами, кроме символьных строк, можно установить другие редакторы ячеек. Это делается аналогично установке редакторов ячеек таблицы, как пояснялось ранее в данной главе.



Рис. 10.30. Редактор ячеек дерева, используемый по умолчанию

В листинге 10.13 приведен весь исходный код рассматриваемого здесь примера программы редактирования отдельных узлов дерева. Запустите эту программу на выполнение, создайте несколько узлов и отредактируйте их, дважды щелкнув на имени узла. Убедитесь в том, что свернутые деревья разворачиваются и содержимое окна прокручивается для отображения нового дочернего узла в области просмотра.

Листинг 10.13. Исходный код из файла `treeEdit/TreeEditFrame.java`

```

1  package treeEdit;
2
3  import java.awt.*;
4
5  import javax.swing.*;
6  import javax.swing.tree.*;
7
8  /**
9   * Фрейм с деревом и кнопками для его редактирования
10  */
11 public class TreeEditFrame extends JFrame
12 {
13     private static final int DEFAULT_WIDTH = 400;
14     private static final int DEFAULT_HEIGHT = 200;
15
16     private DefaultTreeModel model;
17     private JTree tree;
18
19     public TreeEditFrame()
20     {
21         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
22
23         // построить дерево
24
25         TreeNode root = makeSampleTree();
26         model = new DefaultTreeModel(root);
27         tree = new JTree(model);

```

```
28     tree.setEditable(true);
29
30     // ввести прокручиваемую панель с деревом
31
32     JScrollPane scrollPane = new JScrollPane(tree);
33     add(scrollPane, BorderLayout.CENTER);
34
35     makeButtons();
36 }
37
38 public TreeNode makeSampleTree()
39 {
40     DefaultMutableTreeNode root =
41         new DefaultMutableTreeNode("World");
42     DefaultMutableTreeNode country =
43         new DefaultMutableTreeNode("USA");
44     root.add(country);
45     DefaultMutableTreeNode state =
46         new DefaultMutableTreeNode("California");
47     country.add(state);
48     DefaultMutableTreeNode city =
49         new DefaultMutableTreeNode("San Jose");
50     state.add(city);
51     city = new DefaultMutableTreeNode("San Diego");
52     state.add(city);
53     state = new DefaultMutableTreeNode("Michigan");
54     country.add(state);
55     city = new DefaultMutableTreeNode("Ann Arbor");
56     state.add(city);
57     country = new DefaultMutableTreeNode("Germany");
58     root.add(country);
59     state = new DefaultMutableTreeNode("Schleswig-Holstein");
60     country.add(state);
61     city = new DefaultMutableTreeNode("Kiel");
62     state.add(city);
63     return root;
64 }
65
66 /**
67 * Создает кнопки для ввода родственных,
68 * дочерних узлов и их удаления
69 */
70 public void makeButtons()
71 {
72     JPanel panel = new JPanel();
73     JButton addSiblingButton = new JButton("Add Sibling");
74     addSiblingButton.addActionListener(event ->
75     {
76         DefaultMutableTreeNode selectedNode =
77             (DefaultMutableTreeNode) tree
78             .getLastSelectedPathComponent();
79
80         if (selectedNode == null) return;
81
82         DefaultMutableTreeNode parent =
83             (DefaultMutableTreeNode) selectedNode.getParent();
```

```
85         if (parent == null) return;
86
87         DefaultMutableTreeNode newNode =
88             new DefaultMutableTreeNode("New");
89
90         int selectedIndex = parent.getIndex(selectedNode);
91         model.insertNodeInto(newNode, parent, selectedIndex + 1);
92
93         // отобразить теперь новый узел
94
95         TreeNode[] nodes = model.getPathToRoot(newNode);
96         TreePath path = new TreePath(nodes);
97         tree.scrollPathToVisible(path);
98     });
99     panel.add(addSiblingButton);
100
101 JButton addChildButton = new JButton("Add Child");
102 addChildButton.addActionListener(event ->
103 {
104     DefaultMutableTreeNode selectedNode =
105         (DefaultMutableTreeNode) tree
106             .getLastSelectedPathComponent();
107
108     if (selectedNode == null) return;
109
110     DefaultMutableTreeNode newNode =
111         new DefaultMutableTreeNode("New");
112     model.insertNodeInto(newNode, selectedNode,
113         selectedNode.getChildCount());
114
115     // отобразить теперь новый узел
116
117     TreeNode[] nodes = model.getPathToRoot(newNode);
118     TreePath path = new TreePath(nodes);
119     tree.scrollPathToVisible(path);
120 });
121 panel.add(addChildButton);
122
123 JButton deleteButton = new JButton("Delete");
124 deleteButton.addActionListener(event ->
125 {
126     DefaultMutableTreeNode selectedNode =
127         (DefaultMutableTreeNode) tree
128             .getLastSelectedPathComponent();
129
130     if (selectedNode != null
131         && selectedNode.getParent() != null)
132         model.removeNodeFromParent(selectedNode);
133     });
134 panel.add(deleteButton);
135 add(panel, BorderLayout.SOUTH);
136 }
137 }
```

javax.swing.JTree 1.2

- **TreePath getSelectionPath()**

Получает путь к текущему выбранному узлу (или к первому выбранному узлу, если выбрано сразу несколько узлов). Возвращает пустое значение `null`, если ни один из узлов не выбран.

- **Object getLastSelectedPathComponent()**

Получает объект, который представляет текущий выбранный узел (или первый выбранный узел, если выбрано сразу несколько узлов). Возвращает пустое значение `null`, если ни один из узлов не выбран.

- **void makeVisible(TreePath path)**

Развертывает все узлы по заданному пути.

- **void scrollPathToVisible(TreePath path)**

Развертывает все узлы по заданному пути и, если дерево находится на прокручиваемой панели, то прокручивает дерево, чтобы обеспечить отображение последнего узла по заданному пути.

javax.swing.tree.TreePath 1.2

- **Object getLastPathComponent()**

Получает последний объект по заданному пути.

javax.swing.tree.TreeNode 1.2

- **TreeNode getParent()**

Возвращает родительский узел данного узла.

- **TreeNode getChildAt(int index)**

Ищет дочерний узел по указанному индексу. Значение индекса должно находиться в пределах от 0 до `getChildCount() - 1`.

- **int getChildCount()**

Возвращает количество дочерних узлов данного узла.

- **Enumeration children()**

Возвращает объект типа `Enumeration` для перебора всех дочерних узлов данного узла.

javax.swing.tree.DefaultTreeModel 1.2

- **void insertNodeInto(MutableTreeNode newChild, MutableTreeNode parent, int index)**

Вводит объект `newChild` в качестве дочернего узла в родительский узел `parent` по указанному индексу и уведомляет приемники событий от модели дерева.

- **void removeNodeFromParent(MutableTreeNode node)**

Удаляет узел `node` из модели дерева и уведомляет приемники событий от этой модели.

- **void nodeChanged(TreeNode node)**

Уведомляет приемники событий от модели дерева об изменениях в заданном узле `node`.

javax.swing.tree.DefaultTreeModel 1.2 (окончание)

- **void nodesChanged(TreeNode parent, int[] changedChildIndexes)**
Уведомляет приемники событий от модели дерева об изменениях во всех дочерних узлах родительского узла *parent* по указанным индексам.
- **void reload()**
Перезагружает все узлы в модели дерева. Эту операцию нужно выполнять только в тех случаях, когда узлы полностью изменились под внешним воздействием.

10.3.3. Перечисление узлов дерева

Иногда требуется найти узел дерева, начиная с корневого узла и перебирая все дочерние узлы до тех пор, пока не будет найден совпадающий узел. Для перебора узлов в классе DefaultMutableTreeNode предоставляется несколько удобных методов.

Методы *breadthFirstEnumeration()* и *depthFirstEnumeration()* возвращают объект типа Enumeration. У этого объекта имеется метод *nextElement()*, предназначенный для последовательного обращения ко всем дочерним узлам текущего узла. Если объект типа Enumeration получен с помощью метода *breadthFirstEnumeration()*, он представляет результаты обхода в ширину всех дочерних узлов текущего узла. А объект типа Enumeration, возвращаемый методом *depthFirstEnumeration()*, содержит результаты обхода всех дочерних узлов текущего узла в глубину. На рис. 10.31 схематически показаны оба способа обхода узлов дерева.

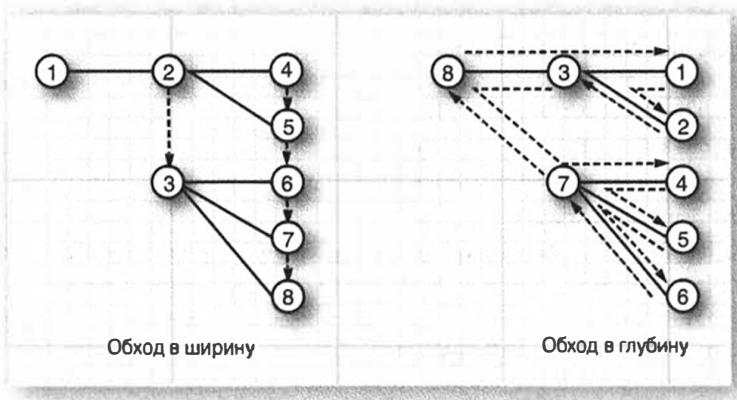


Рис. 10.31. Способы обхода узлов дерева

Обход узлов дерева в ширину выполняется по отдельным уровням: сначала корневой узел, затем все его дочерние узлы, после чего все дочерние узлы этих дочерних узлов и т.д. Воспроизвести результаты такого обхода совсем не трудно.

А вот обход узлов дерева в глубину похож на поиск выхода из лабиринта. В этом случае обход выполняется по какому-то одному пути до тех пор, пока не

будет достигнут листовой узел. Затем происходит возврат назад и выбор ближайшего нового пуги, после чего обход продолжается по этому же пути до тех пор, пока не будет достигнут листовой узел, и т.д.

Такой способ иначе называется *обходом в обратном порядке*, поскольку в процессе поиска сначала посещаются дочерние, а затем родительские узлы. Именно поэтому метод `postOrderTraversal()` действует подобно методу `depthFirstTraversal()`. Следует отметить и метод `preOrderTraversal()`, который также предназначен для обхода в глубину, но в получаемых результатах родительские узлы предшествуют дочерним. Ниже приведен образец типичной реализации обхода дерева в ширину непосредственно в коде.

```
Enumeration breadthFirst = node.breadthFirstEnumeration();
while (breadthFirst.hasMoreElements())
    сделать что-нибудь с результатом вызова
    метода breadthFirst.nextElement();
```

И, наконец, метод `pathFromAncestorEnumeration()` предназначается для поиска пути от родительского узла к заданному и для перечисления узлов по этому пути. Принцип его работы основывается на вызове метода `getParent()` для данного узла до тех пор, пока не будет найден заданный родительский узел. После этого предоставляется путь для обхода дерева обратном порядке.

В следующем примере программы демонстрируется применение описанных выше методов для обхода дерева иерархического наследования классов. Как только в текстовом поле, расположенном в нижней части рабочего окна данной программы, будет указано имя конкретного класса, он будет введен в дерево иерархического наследования со всеми своими суперклассами (рис. 10.32).

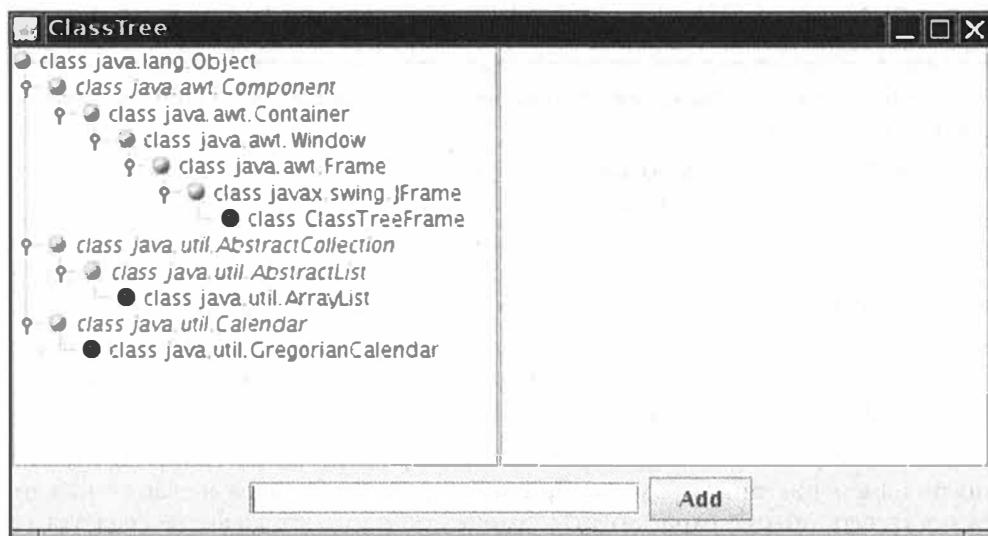


Рис. 10.32. Дерево иерархического наследования классов

В рассматриваемом здесь примере используется следующая важная особенность дерева: пользовательский объект узла может быть объектом произвольного типа. В данном примере узлы дерева представляют иерархическую структуру

наследования классов, поэтому они относятся к одному типу Class. Во избежание дублирования следует проверить наличие добавляемого класса в дереве с помощью приведенного ниже метода.

```
public DefaultMutableTreeNode findUserObject(Object obj)
{
    Enumeration e = root.breadthFirstEnumeration();
    while (e.hasMoreElements())
    {
        DefaultMutableTreeNode node =
            (DefaultMutableTreeNode) e.nextElement();
        if (node.getUserObject().equals(obj))
            return node;
    }
    return null;
}
```

10.3.4. Воспроизведение узлов дерева

При разработке прикладных программ нередко возникает потребность изменить способ отображения узлов дерева, например, обозначить их другими пиктограммами или выделить их названия другим шрифтом. Подобные изменения можно вносить с помощью *средства воспроизведения ячеек дерева*. По умолчанию для воспроизведения узлов дерева в компоненте JTree применяется класс DefaultTreeCellRenderer, который расширяет класс JLabel. Этот класс формирует метку, состоящую из пиктограммы и метки узла.



На заметку! Средство воспроизведения ячеек дерева не отображает маркеры свертывания и развертывания подчиненных деревьев. Они являются частью общего визуального стиля, поэтому изменять их не рекомендуется.

Специальную настройку воспроизведения узлов дерева можно произвести одним из трех способов.

- Заменить с помощью класса DefaultTreeCellRenderer пиктограммы, шрифты и цвета фона, применяемые во всех узлах дерева.
- Установить используемое по умолчанию средство воспроизведения, расширяющее класс DefaultTreeCellRenderer, а также применить разные пиктограммы, шрифты и цвета фона в отдельных узлах дерева.
- Установить средство воспроизведения, реализующее интерфейс TreeCellRenderer, чтобы воспроизводить в отдельных узлах дерева специально предусмотренные для них изображения.

Рассмотрим каждый из этих способов более подробно. Наиболее простой способ изменения пиктограммы, шрифта и цвета фона узлов состоит в том, чтобы построить объект типа DefaultTreeCellRenderer в качестве средства воспроизведения узлов дерева по умолчанию и установить его вместе с требуемыми пиктограммами в дереве, как показано ниже. Результат выполнения этих действий приведен на рис. 10.32, где в качестве пиктограмм узлов используются изображения шариков.

```

DefaultTreeCellRenderer renderer = new DefaultTreeCellRenderer();
renderer.setLeafIcon(new ImageIcon("blue-ball.gif"));
    // используется для листовых узлов
renderer.setClosedIcon(new ImageIcon("red-ball.gif"));
    // используется для свернутых узлов
renderer.setOpenIcon(new ImageIcon("yellow-ball.gif"));
    // используется для развернутых узлов
tree.setCellRenderer(renderer);

```

Изменять шрифт и цвет фона не рекомендуется, так как легко нарушить общий стиль оформления ГПИ. Изменять шрифт допускается только для выделения отдельных узлов. Так, на рис. 6.32 курсивом выделены абстрактные классы.

Для изменения внешнего вида отдельных узлов следует установить специальное средство воспроизведения ячеек дерева, во многом схоже со средством воспроизведения ячеек из списка, рассмотренным ранее в этой главе. У интерфейса TreeCellRenderer имеется следующий единственный метод getTreeCellRendererComponent():

```

Component getTreeCellRendererComponent(JTree tree, Object value,
    boolean selected, boolean expanded, boolean leaf,
    int row, boolean hasFocus)

```

Метод getTreeCellRendererComponent() из класса DefaultTreeCellRenderer возвращает ссылку this, т.е. текущую метку. (Напомним, что класс DefaultTreeCellRenderer расширяет класс JLabel.) Для создания специального компонента, предназначенного для воспроизведения ячеек дерева, необходимо сначала расширить класс DefaultTreeCellRenderer. Затем следует переопределить метод getTreeCellRendererComponent(), предусмотрев в нем вызов из суперкласса метода, который подготовит все данные, необходимые для создания метки. Кроме того, в данном методе задаются требующиеся значения свойств, а по завершении своего выполнения он возвращает ссылку this.

```

class MyTreeCellRenderer extends DefaultTreeCellRenderer
{
    public Component getTreeCellRendererComponent(JTree tree,
        Object value, boolean selected, boolean expanded,
        boolean leaf, int row, boolean hasFocus)
    {
        Component comp =
            super.getTreeCellRendererComponent(tree, value, selected,
                expanded, leaf, row, hasFocus);
        DefaultMutableTreeNode node = (DefaultMutableTreeNode) value;
        см. пользовательский объект node.getUserObject();
        Font font = подходящий шрифт;
        comp.setFont(font);
        return comp;
    }
};

```

Внимание! Параметр value метода getTreeCellRendererComponent() является узловым, а не пользовательским объектом. Напомним, что пользовательский объект относится к классу узлов DefaultMutableTreeNode, а класс JTree может содержать узлы произвольного типа. Если в дереве используются узлы типа DefaultMutableTreeNode, то на второй стадии следует извлечь пользовательский объект, как это было сделано в предыдущем примере кода.



Внимание! В классе `DefaultTreeCellRenderer` используется один и тот же объект метки для всех узлов, но для каждого узла изменяется ее текст. Если изменить шрифт для выделения названия отдельного узла, то при последующем вызове упомянутого выше метода следует восстановить используемый по умолчанию шрифт. В противном случае названия всех последующих узлов будут выделены новым шрифтом! Один из способов восстановления исходного шрифта представлен далее, в листинге 20.14.

Здесь не приводится пример, демонстрирующий применение средства воспроизведения ячеек дерева для отображения произвольной графики. Если у вас возникнет потребность в этом, можете без особого труда приспособить под свои нужды средство воспроизведения ячеек из списка, представленное в листинге 10.4.

Средство воспроизведения типа `ClassNameTreeCellRenderer` из листинга 10.14 выделяет имя класса обычным шрифтом или курсивом в зависимости от наличия модификатора `ABSTRACT` у объекта типа `Class`. Для обозначения абстрактных классов не выбирается какой-то другой шрифт, чтобы не изменять визуальный стиль, обычно применяемый для отображения дерева. По этой причине курсив для обозначения абстрактного класса получается путем наклона начертания исходного шрифта, которым выделяются метки. Напомним, что в результате всех вызовов возвращается только один общий объект типа `JLabel`. При последующих вызовах метода `getTreeCellRendererComponent()` необходимо снова вернуться к исходному шрифту. Обратите также внимание на то, каким образом изменяются пиктограммы узлов в конструкторе класса `ClassTreeFrame`.

`javax.swing.tree.DefaultMutableTreeNode` 1.2

- `Enumeration breadthFirstEnumeration()`
- `Enumeration depthFirstEnumeration()`
- `Enumeration preOrderEnumeration()`
- `Enumeration postOrderEnumeration()`

Возвращают объект типа `Enumeration`, представляющий результаты обхода всех узлов в модели дерева. При обходе в ширину дочерние узлы, находящиеся ближе к корневому узлу, посещаются раньше. При обходе в глубину перед переходом к равноправному узлу посещаются все дочерние узлы. Метод `postOrderEnumeration()` является аналогом метода `depthFirstEnumeration()`. Обход в ширину [или в прямом порядке] подобен обходу в глубину [или в обратном порядке] за исключением того, что родительские узлы перечисляются раньше дочерних.

`javax.swing.tree.TreeCellRenderer` 1.2

- `Component getTreeCellRendererComponent(JTree tree, Object value, boolean selected, boolean expanded, boolean leaf, int row, boolean hasFocus)`

Возвращает компонент, метод `paint()` которого вызывается для воспроизведения ячейки [или узла] дерева.

javax.swing.tree.TreeCellRenderer 1.2 (окончание)

Параметры:	tree	Дерево, содержащее воспроизведимый узел
	value	Воспроизведимый узел
	selected	Принимает логическое значение
		true , если выбран данный узел
	expanded	Принимает логическое значение
		true , если видны дочерние узлы данного узла
	leaf	Принимает логическое значение
		true , если данный узел должен быть отображен как лист
	row	Отображаемая строка, содержащая узел
	hasFocus	Принимает логическое значение
		true , если данный узел обладает фокусом ввода

javax.swing.tree.DefaultTreeCellRenderer 1.2

- **void setLeafIcon(Icon icon)**
- **void setOpenIcon(Icon icon)**
- **void setClosedIcon(Icon icon)**

Задают пиктограмму для обозначения листового, развернутого или свернутого узла.

10.3.5. Обработка событий в деревьях

Дерево чаще всего используется совместно с каким-нибудь другим компонентом. Например, при выборе какого-нибудь узла дерева в соседнем окне может быть показана определенная сопроводительная информация. На рис. 10.33 представлен пример программы, где для каждого узла в правом текстовом окне автоматически отображаются статические переменные и переменные экземпляра соответствующего класса.

Чтобы добиться такого поведения, необходимо установить *приемник событий выбора из дерева*. Класс такого приемника событий должен реализовать интерфейс TreeSelectionListener со следующим единственным методом valueChanged():

```
void valueChanged(TreeSelectionEvent event)
```

Метод valueChanged() вызывается при каждом событии выбора или отмены выбора узлов дерева. Приемник событий добавляется к дереву обычным образом:

```
tree.addTreeSelectionListener(listener);
```

С помощью свойств модели типа TreeSelectionModel можно специаль- но указать конкретный режим выбора узлов дерева: только один узел (свойство

SINGLE_TREE_SELECTION), группа смежных узлов (свойство CONTINGUOUS_TREE_SELECTION) или произвольная группа несмежных узлов (свойство DISCONTINGUOUS_TREE_SELECTION). По умолчанию допускается выбор произвольной группы несмежных узлов. В рассматриваемом здесь примере программы для просмотра классов допускается выбор только одного узла дерева, как показано ниже. Никаких других действий, кроме указания конкретного режима выбора, предпринимать не нужно.

```
int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
tree.getSelectionModel().setSelectionMode(mode);
```

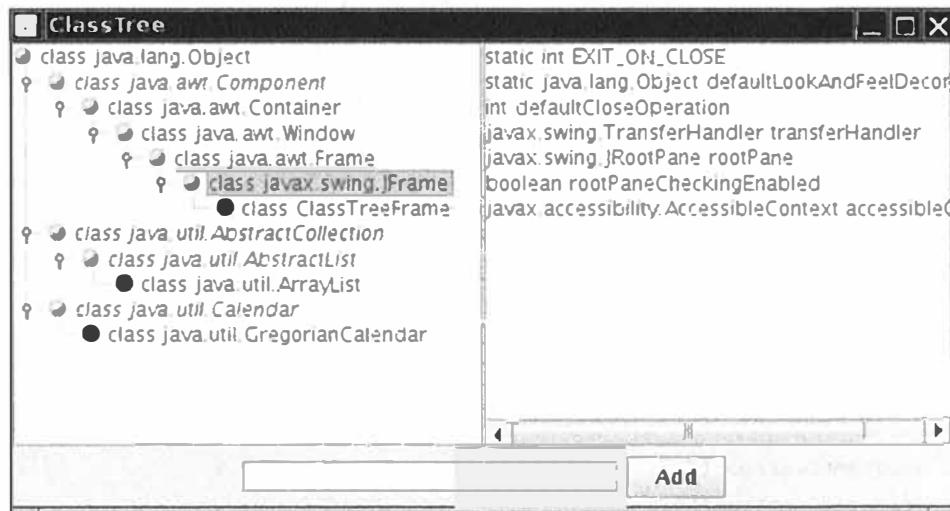


Рис. 10.33. Пример программы для просмотра классов

На заметку! Порядок выбора нескольких элементов зависит от конкретного визуального стиля ГПИ. Так, если применяется визуальный стиль Metal, для выбора несмежных элементов [в данном случае узлов дерева] следует нажать клавишу <Ctrl> и, не отпуская ее, щелкнуть по очереди на каждом выбиряемом элементе, а для выбора нескольких смежных элементов — нажать клавишу <Shift> и, не отпуская ее, щелкнуть сначала на первом, а затем на последнем элементе из выбираемой группы.

Чтобы выяснить, что именно выбрано в настоящий момент, необходимо запросить дерево, вызвав метод `getSelectionPaths()` следующим образом:

```
TreePath[] selectedPaths = tree.getSelectionPaths();
```

Если же требуется ограничить возможности пользователя, разрешив ему выбирать узлы дерева только по одному, то для этой цели удобнее воспользоваться служебным методом `getSelectionPath()`, который в общем случае возвращает первый же выбранный путь к элементу или пустое значение `null`, если такой путь не выбран.

Внимание! В классе `TreeSelectionEvent` имеется метод `getPaths()`, который возвращает массив объектов типа `TreePath`, но этот массив описывает изменения в самом выборе, а не текущий результат выбора.

В листинге 10.14 приведен исходный код класса фрейма для программы, отображающей иерархию наследования классов в виде древовидной структуры, где абстрактные классы выделены курсивом. А в листинге 10.15 представлен исходный код, реализующий средство воспроизведения ячеек дерева. Если ввести полное имя класса и нажать клавишу <Enter> или щелкнуть на кнопке Add, в дерево будет введен новый класс со всеми его суперклассами. В полное имя класса следует включить имя пакета, как, например, `java.util.ArrayList`.

Рассматриваемая здесь программа несколько сложна для понимания, поскольку в ней для построения дерева классов используется механизм рефлексии. Весь код построения дерева классов находится в методе `addClass()`. (Иерархия наследования классов используется здесь лишь в качестве удобного примера для демонстрации особенностей построения деревьев и обращения с ними, не усложняя программирование. При разработке собственных прикладных программ вы можете использовать любые другие источники иерархических данных.) Для проверки наличия вводимого класса в дереве вызывается метод `findUserObject()`, реализация которого описана в предыдущем разделе. В этом методе реализуется алгоритм обхода дерева в ширину. Если класс отсутствует в дереве, то сначала вводятся его суперклассы, а затем создается и отображается новый узел для данного класса.

Когда выбран узел дерева, текстовая область справа заполняется полями выбранного класса. В конструкторе фрейма накладываются ограничения, разрешающие пользователю данной программы выбирать узлы дерева только по одному, а также вводится приемник событий выбора из дерева. При вызове метода `valueChanged()` его параметр события игнорируется, а у дерева запрашивается только путь к текущему выбранному узлу. Далее последний узел, как обычно, получается по заданному пути, и из него извлекается пользовательский объект. После этого вызывается метод `getFieldDescription()`, где для формирования символьной строки со всеми полями выбранного класса применяется механизм рефлексии.

Листинг 10.14. Исходный код из файла `treeRender/ClassTreeFrame.java`

```
1 package treeRender;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.lang.reflect.*;
6 import java.util.*;
7
8 import javax.swing.*;
9 import javax.swing.tree.*;
10
11 /**
12 * В этом фрейме отображается дерево иерархии классов,
13 * текстовое поле и кнопка Add для ввода новых классов в дерево
14 */
15 public class ClassTreeFrame extends JFrame
16 {
17     private static final int DEFAULT_WIDTH = 400;
18     private static final int DEFAULT_HEIGHT = 300;
19
20     private DefaultMutableTreeNode root;
21     private DefaultTreeModel model;
```

```
22     private JTree tree;
23     private JTextField textField;
24     private JTextArea textArea;
25
26     public ClassTreeFrame()
27     {
28         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
29
30         // в корне дерева иерархии классов находится класс Object
31         root = new DefaultMutableTreeNode(java.lang.Object.class);
32         model = new DefaultTreeModel(root);
33         tree = new JTree(model);
34
35         // ввести этот класс, чтобы заполнить дерево рядом данных
36         addClass(getClass());
37
38         // установить пиктограммы для обозначения узлов
39         ClassNameTreeCellRenderer renderer =
40             new ClassNameTreeCellRenderer();
41         renderer.setClosedIcon(new ImageIcon(getClass()
42             .getResource("red-ball.gif")));
43         renderer.setOpenIcon(new ImageIcon(getClass()
44             .getResource("yellow-ball.gif")));
45         renderer.setLeafIcon(new ImageIcon(getClass()
46             .getResource("blue-ball.gif")));
47         tree.setCellRenderer(renderer);
48
49         // установить режим выбора узлов дерева
50         tree.addTreeSelectionListener(event ->
51         {
52             // пользователь выбрал другой узел -
53             // обновить его описание
54             TreePath path = tree.getSelectionPath();
55             if (path == null) return;
56             DefaultMutableTreeNode selectedNode =
57                 (DefaultMutableTreeNode) path
58                 .getLastPathComponent();
59             Class<?> c = (Class<?>) selectedNode.getUserObject();
60             String description = getFieldDescription(c);
61             textArea.setText(description);
62         });
63         int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
64         tree.getSelectionModel().setSelectionMode(mode);
65
66         // в этой текстовой области находится описание класса
67         textArea = new JTextArea();
68
69         // добавить дерево и текстовую область
70         JPanel panel = new JPanel();
71         panel.setLayout(new GridLayout(1, 2));
72         panel.add(new JScrollPane(tree));
73         panel.add(new JScrollPane(textArea));
74
75         add(panel, BorderLayout.CENTER);
76
77         addTextField();
78     }
79
80     /**
81      * Добавляет в фрейм текстовое поле и кнопку Add
82  
```

```
82     * для ввода нового класса
83     */
84     public void addTextField()
85     {
86         JPanel panel = new JPanel();
87
88         ActionListener addListener = event ->
89         {
90             // ввести класс, имя которого находится в текстовом поле
91             try
92             {
93                 String text = textField.getText();
94                 addClass(Class.forName(text));
95                 // очистить текстовое поле, чтобы обозначить
96                 // удачный исход ввода класса
97                 textField.setText("");
98             }
99             catch (ClassNotFoundException e)
100             {
101                 JOptionPane.showMessageDialog(null,
102                                         "Class not found");
103             }
104         };
105
106         // в этом текстовом поле вводятся имена новых классов
107         textField = new JTextField(20);
108         textField.addActionListener(addListener);
109         panel.add(textField);
110
111         JButton addButton = new JButton("Add");
112         addButton.addActionListener(addListener);
113         panel.add(addButton);
114
115         add(panel, BorderLayout.SOUTH);
116     }
117
118 /**
119 * Находит искомый объект в дереве
120 * @param obj Искомый объект
121 * @return Возвращает узел, содержащий объект, или пустое
122 *         значение null, если объект отсутствует в дереве
123 */
124 @SuppressWarnings("unchecked")
125 public DefaultMutableTreeNode findUserObject(Object obj)
126 {
127     // найти узел, содержащий пользовательский объект
128     Enumeration<TreeNode> e =
129         (Enumeration<TreeNode>) root.breadthFirstEnumeration();
130     while (e.hasMoreElements())
131     {
132         DefaultMutableTreeNode node =
133             (DefaultMutableTreeNode) e.nextElement();
134         if (node.getUserObject().equals(obj)) return node;
135     }
136     return null;
137 }
138
139 /**
140 * Вводит новый класс и любые его родительские классы,
141 * пока еще отсутствующие в дереве
```

```
142     * @param с Вводимый класс
143     * @return Возвращает узел с вновь введенным классом
144     */
145    public DefaultMutableTreeNode addClass(Class<?> c)
146    {
147        // ввести новый класс в дерево
148
149        // пропустить типы данных, не относящиеся к классам
150        if (c.isInterface() || c.isPrimitive()) return null;
151
152        // если класс уже присутствует в дереве, возвратить его узел
153        DefaultMutableTreeNode node = findUserObject(c);
154        if (node != null) return node;
155
156        // класс отсутствует в дереве – ввести сначала его
157        // родительские классы рекурсивным способом
158
159        Class<?> s = c.getSuperclass();
160
161        DefaultMutableTreeNode parent;
162        if (s == null) parent = root;
163        else parent = addClass(s);
164
165        // ввести затем класс как потомок его родительского класса
166        DefaultMutableTreeNode newNode =
167            new DefaultMutableTreeNode(c);
168        model.insertNodeInto(newNode, parent, parent.getChildCount());
169
170        // сделать видимым узел с вновь введенным классом
171        TreePath path = new TreePath(model.getPathToRoot(newNode));
172        tree.makeVisible(path);
173
174        return newNode;
175    }
176
177 /**
178 * Возвращает описание полей класса
179 * @param Описываемый класс
180 * @return Возвращает символьную строку, содержащую все
181 *         типы и имена полей описываемого класса
182 */
183    public static String getFieldDescription(Class<?> c)
184    {
185        // использовать рефлексию для обнаружения типов и имен полей
186        StringBuilder r = new StringBuilder();
187        Field[] fields = c.getDeclaredFields();
188        for (int i = 0; i < fields.length; i++)
189        {
190            Field f = fields[i];
191            if ((f.getModifiers() & Modifier.STATIC) != 0)
192                r.append("static ");
193            r.append(f.getType().getName());
194            r.append(" ");
195            r.append(f.getName());
196            r.append("\n");
197        }
198        return r.toString();
199    }
200 }
```

Листинг 10.15. Исходный код из файла treeRender/ClassNameTreeCellRenderer.java

```
1 package treeRender;
2
3 import java.awt.*;
4 import java.lang.reflect.*;
5 import javax.swing.*;
6 import javax.swing.tree.*;
7 /**
8  * Этот класс воспроизводит имя класса, выделяя его
9  * простым шрифтом или курсивом. Абстрактные классы
10 * выделяются только курсивом
11 */
12 public class ClassNameTreeCellRenderer extends
13         DefaultTreeCellRenderer
14 {
15     private Font plainFont = null;
16     private Font italicFont = null;
17
18     public Component getTreeCellRendererComponent(
19             JTree tree, Object value, boolean selected,
20             boolean expanded, boolean leaf, int row,
21             boolean hasFocus)
22     {
23         super.getTreeCellRendererComponent(
24             tree, value, selected, expanded, leaf, row, hasFocus);
25         // получить пользовательский объект
26         DefaultMutableTreeNode node = (DefaultMutableTreeNode) value;
27         Class<?> c = (Class<?>) node.getUserObject();
28
29         // сначала сделать простой шрифт наклонным
30         if (plainFont == null)
31         {
32             plainFont = getFont();
33             // средство воспроизведения ячеек дерева иногда
34             // вызывается с меткой, имеющей пустой шрифт
35             if (plainFont != null) italicFont =
36                 plainFont.deriveFont(Font.ITALIC);
37         }
38
39         // установить наклонный шрифт, если класс является
40         // абстрактным, а иначе – простой шрифт
41         if ((c.getModifiers() & Modifier.ABSTRACT) == 0)
42             setFont(plainFont);
43         else setFont(italicFont);
44         return this;
45     }
46 }
```

javax.swing.JTree 1.2

- **TreePath getSelectionPath()**
- **TreePath[] getSelectionPaths()**

Возвращают путь к первому выбранному узлу или массив путей ко всем выбранным узлам дерева. Если ни один из узлов дерева не выбран, оба метода возвращают пустое значение **null**.

javax.swing.event.TreeSelectionListener 1.2

- **void valueChanged(TreeSelectionEvent event)**

Вызывается всякий раз, когда происходит выбор или отмена выбора узлов дерева.

javax.swing.event.TreeSelectionEvent 1.2

- **TreePath getPath()**
- **TreePath[] getPaths()**

Получают первый путь или все пути, которые изменились в результате данного события выбора из дерева. Для получения сведений о текущем выборе, а не об изменении выбора следует вызывать метод `JTree.getSelectionPath()`.

10.3.6. Специальные модели деревьев

В качестве последнего примера манипулирования деревьями рассмотрим программу, которая, подобно отладчику, проверяет содержимое переменной или объекта. На рис. 10.34 показано рабочее окно данной программы.

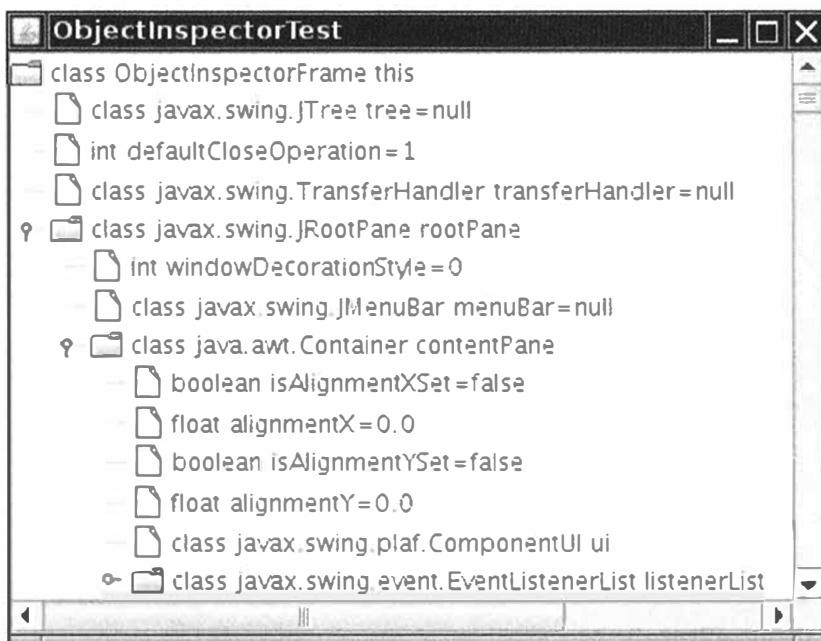


Рис. 10.34. Пример дерева для обследования объектов

Скомпилируйте и запустите эту программу на выполнение, прежде чем читать ее описание. Каждый узел дерева соответствует какому-нибудь полю экземпляра. Если поле содержит объект, разверните структуру этого объекта для просмотра полей экземпляра его класса. Как видите, данная программа позволяет

просматривать содержимое своего фрейма. При более внимательном изучении нескольких полей экземпляра вы можете обнаружить некоторые уже знакомые вам классы. Кроме того, эта программа позволяет оценить, насколько сложны компоненты библиотеки Swing, предназначенные для построения ГПИ.

Данная программа примечательна тем, что для построения дерева в ней не применяется модель типа DefaultTreeModel. При наличии данных, которые уже иерархически организованы в виде древовидной структуры, вряд ли имеет смысл использовать предлагаемое по умолчанию дерево и синхронизировать оба дерева. Именно такая ситуация возникает в рассматриваемом здесь примере: обследуемые объекты уже связаны друг с другом с помощью ссылок на них, и поэтому нет никакой нужды дублировать структуру их связей.

В интерфейсе TreeModel объявлено лишь несколько удобных методов. Первая группа методов позволяет найти узлы дерева, начиная с корня и продолжая дочерними узлами. Эти методы вызываются в классе JTree только в том случае, если узел был развернут пользователем, как показано ниже.

```
Object getRoot()  
int getChildCount(Object parent)  
Object getChild(Object parent, int index)
```

В рассматриваемом здесь примере наглядно демонстрируется, почему для интерфейса TreeModel, как и для класса JTree, не требуется явное обозначение узлов. Корневые и дочерние узлы могут быть объектами произвольного типа, интерфейс TreeModel отвечает за уведомление класса JTree об установлении связи между ними. Следующий метод выполняет действия, обратные методу getChild():

```
int getIndexOfChild(Object parent, Object child)
```

Как следует из листинга 10.16, этот метод можно реализовать на основе первых трех методов. Модель дерева сообщает компоненту JTree, какие именно узлы следует отобразить как листовые:

```
boolean isLeaf(Object node)
```

Если при выполнении кода изменяется модель дерева, то дерево следует уведомить о необходимости его перерисовки. Поэтому дерево вводится в модель как приемник событий типа TreeModelListener. Следовательно, в модели должны использоваться обычные методы управления обработчиками событий, как показано ниже, а их реализация представлена в листинге 10.17.

```
void addTreeModelListener(TreeModelListener l)  
void removeTreeModelListener(TreeModelListener l)
```

Для изменения содержимого дерева в его модели вызывается один из четырех перечисленных ниже методов из интерфейса TreeModelListener.

```
void treeNodesChanged(TreeModelEvent e)  
void treeNodesInserted(TreeModelEvent e)  
void treeNodesRemoved(TreeModelEvent e)  
void treeStructureChanged(TreeModelEvent e)
```

Объект типа TreeModelEvent описывает место, где происходят изменения в дереве. Подробности организации событий, наступающих в модели дерева при вводе и удалении узлов, здесь не рассматриваются, поскольку они носят слишком

технический характер. Вам нужно лишь позаботиться об инициировании подобных событий, когда в дереве вводятся и удаляются отдельные узлы. В листинге 10.16 демонстрируется пример инициирования события, наступающего при замене корневого узла новым объектом.



Совет! Для упрощения кода инициирования событий рекомендуется удобный класс `java.awt.event.ActionListener`, предназначенный для составления списка из приемников событий. На примере трех последних методов из листинга 10.17 показано, как пользоваться этим классом в прикладном коде.

И наконец, если пользователь редактирует узел дерева, то его модель вызывается с помощью следующего метода, где указывается вносимое изменение:

```
void valueForPathChanged(TreePath path, Object newValue)
```

Если же редактирование узлов не разрешается, этот метод вообще не вызывается. В таком случае создать модель еще проще. Для этого достаточно реализовать три метода:

```
Object getRoot()  
int getChildCount(Object parent)  
Object getChild(Object parent, int index)
```

Эти методы описывают структуру дерева. После реализации остальных пяти методов, как показано в листинге 10.16, можно приступать к отображению дерева.

А теперь перейдем непосредственно к реализации рассматриваемого здесь примера программы. В ней строится дерево, состоящее из объектов типа `Variable` в его узлах.



На заметку! Если бы в данном примере применялась модель дерева типа `DefaultTreeModel`, то узлы были бы объектами типа `DefaultMutableTreeNode` с пользовательскими объектами типа `Variable`.

Допустим, требуется проверить переменную, объявленную следующим образом:

```
Employee joe;
```

Эта переменная имеет тип `Employee.class`, имя `joe` и значение в виде ссылки на объект `joe`. В листинге 10.18 класс `Variable`, описывающий эту переменную в рассматриваемом здесь примере программы, определяется следующим образом:

```
Variable v = new Variable(Employee.class, "joe", joe);
```

Если переменная относится к примитивному типу, то для ее значения придется создать объектную оболочку:

```
new Variable(double.class, "salary", new Double(salary));
```

Если же переменная относится к типу класса, то она имеет поля, которые можно перечислить и собрать в объекте типа `ArrayList` с помощью механизма рефлексии. Метод `getFields()` из класса `Class` не возвращает поля суперкласса, поэтому данный метод придется вызвать для всех суперклассов проверяемого класса. Для этой цели служит исходный код, находящийся в конструкторе класса `Variable`. Метод `getFields()` из класса `Variable` возвращает массив полей.

И наконец, метод `toString()` из класса `Variable` форматирует метку узла дерева. Метка всегда содержит тип и имя переменной, а если переменная не относится к типу класса, то метка содержит также ее значение.



На заметку! Если тип переменной представлен массивом, то его элементы в данном примере не отображаются. Впрочем, сделать это совсем не трудно, и такая возможность предоставляется читателям в качестве дополнительного упражнения.

Перейдем непосредственно к модели дерева. Исходный код первых двух ее методов несложен:

```
public Object getRoot()
{
    return root;
}

public int getChildCount(Object parent)
{
    return ((Variable) parent).getFields().size();
}
```

Метод `getChild()` возвращает новый объект типа `Variable`, описывающий поле по указанному индексу. С помощью методов `getType()` и `getName()` из класса `Field` можно получить тип и имя поля, а с помощью механизма рефлексии — прочитать значение поля, вызвав метод `f.get(parentValue)`. Этот метод может генерировать исключение типа `IllegalAccessException`. Но в конструкторе класса `Variable` предоставлен свободный доступ ко всем полям, поэтому подобная исключительная ситуация вряд ли вообще возникнет. Ниже приведен весь исходный код метода `getChild()`.

```
public Object getChild(Object parent, int index)
{
    ArrayList fields = ((Variable) parent).getFields();
    Field f = (Field) fields.get(index);
    Object parentValue = ((Variable) parent).getValue();
    try
    {
        return new Variable(f.getType(), f.getName(), f.get(parentValue));
    }
    catch (IllegalAccessException e)
    {
        return null;
    }
}
```

Упомянутые выше три метода выявляют структуру дерева объектов для компонента `JTree`. А остальные методы выполняют другие рутинные операции, как следует из листинга 10.17.

Необходимо отметить следующую особенность рассматриваемой здесь модели дерева: она описывает бесконечное дерево. Это можно проверить на примере одного из объектов типа `WeakReference`. Так, если щелкнуть в дереве на имени переменной `referent`, произойдет возврат к исходному объекту, который содержит идентичное подчиненное дерево с объектом типа `WeakReference`. Разумеется, сохранить бесконечное количество узлов невозможно, поэтому данная модель

дерева формирует дочерние узлы по требованию в процессе постепенного развертывания родительских узлов. Исходный код класса фрейма для данной программы представлен в листинге 10.16.

Листинг 10.16. Исходный код из файла treeModel/ObjectInspectorFrame.java

```

1 package treeModel;
2
3 import java.awt.*;
4 import javax.swing.*;
5
6 /**
7  * Этот фрейм содержит дерево объектов
8  */
9 public class ObjectInspectorFrame extends JFrame
10 {
11     private JTree tree;
12     private static final int DEFAULT_WIDTH = 400;
13     private static final int DEFAULT_HEIGHT = 300;
14
15     public ObjectInspectorFrame()
16     {
17         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
18
19         // В данном фрейме обследуются объекты
20
21         Variable v = new Variable(getClass(), "this", this);
22         ObjectTreeModel model = new ObjectTreeModel();
23         model.setRoot(v);
24
25         // построить и показать дерево
26
27         tree = new JTree(model);
28         add(new JScrollPane(tree), BorderLayout.CENTER);
29     }
30 }
```

Листинг 10.17. Исходный код из файла treeModel/ObjectTreeModel.java

```

1 package treeModel;
2
3 import java.lang.reflect.*;
4 import java.util.*;
5 import javax.swing.event.*;
6 import javax.swing.tree.*;
7
8 /**
9  * Эта модель дерева описывает древовидную структуру объектов
10 * в Java. Дочерние узлы дерева являются объектами, хранящимися
11 * в переменных экземпляра
12 */
13 public class ObjectTreeModel implements TreeModel
14 {
15     private Variable root;
16     private EventListenerList listenerList = new EventListenerList();
17 }
```

```
18  /**
19   * Строит пустое дерево
20  */
21 public ObjectTreeModel()
22 {
23     root = null;
24 }
25
26 /**
27  * Устанавливает заданную переменную в корне дерева
28  * @param v Переменная, описываемая в данном дереве
29 */
30 public void setRoot(Variable v)
31 {
32     Variable oldRoot = v;
33     root = v;
34     fireTreeStructureChanged(oldRoot);
35 }
36
37 public Object getRoot()
38 {
39     return root;
40 }
41
42 public int getChildCount(Object parent)
43 {
44     return ((Variable) parent).getFields().size();
45 }
46
47 public Object getChild(Object parent, int index)
48 {
49     ArrayList<Field> fields = ((Variable) parent).getFields();
50     Field f = (Field) fields.get(index);
51     Object parentValue = ((Variable) parent).getValue();
52     try
53     {
54         return new Variable(f.getType(), f.getName(),
55                             f.get(parentValue));
56     }
57     catch (IllegalAccessException e)
58     {
59         return null;
60     }
61 }
62
63 public int getIndexOfChild(Object parent, Object child)
64 {
65     int n = getChildCount(parent);
66     for (int i = 0; i < n; i++)
67         if (getChild(parent, i).equals(child)) return i;
68     return -1;
69 }
70
71 public boolean isLeaf(Object node)
72 {
73     return getChildCount(node) == 0;
74 }
75
76 public void valueForPathChanged(TreePath path, Object newValue)
77 {
```

```
78 }
79
80 public void addTreeModelListener(TreeModelListener l)
81 {
82     listenerList.add(TreeModelListener.class, l);
83 }
84
85 public void removeTreeModelListener(TreeModelListener l)
86 {
87     listenerList.remove(TreeModelListener.class, l);
88 }
89
90 protected void fireTreeStructureChanged(Object oldRoot)
91 {
92     TreeModelEvent event =
93         new TreeModelEvent(this, new Object[] { oldRoot });
94     for (TreeModelListener l :
95         listenerList.getListeners(TreeModelListener.class))
96         l.treeStructureChanged(event);
97 }
98 }
```

Листинг 10.18. Исходный код из файла treeModel/Variable.java

```
1 package treeModel;
2
3 import java.lang.reflect.*;
4 import java.util.*;
5
6 /**
7  * Переменная с типом, именем и значением
8 */
9 public class Variable
10 {
11     private Class<?> type;
12     private String name;
13     private Object value;
14     private ArrayList<Field> fields;
15
16 /**
17  * Сконструировать переменную
18  * @param aType Тип
19  * @param aName Имя
20  * @param aValue Значение
21 */
22     public Variable(Class<?> aType, String aName, Object aValue)
23     {
24         type = aType;
25         name = aName;
26         value = aValue;
27         fields = new ArrayList<>();
28
29         // найти все поля, если это тип класса, за исключением
30         // символьных строк и пустых значений null
31
32         if (!type.isPrimitive() && !type.isArray() &&
33             !type.equals(String.class) && value != null)
34         {
35             // получить поля из класса и всех его суперклассов
```

```

36     for (Class<?> c = value.getClass(); c != null;
37         c = c.getSuperclass())
38     {
39         Field[] fs = c.getDeclaredFields();
40         AccessibleObject.setAccessible(fs, true);
41
42         // получить все нестатические поля
43         for (Field f : fs)
44             if ((f.getModifiers() & Modifier.STATIC) == 0)
45                 fields.add(f);
46     }
47 }
48 }
49
50 /**
51 * Получает значение данной переменной
52 * @return Возвращает значение
53 */
54 public Object getValue()
55 {
56     return value;
57 }
58
59 /**
60 * Получает все нестатические поля из данной переменной
61 * @return Возвращает списочный массив переменных
62 *         с описаниями полей
63 */
64 public ArrayList<Field> getFields()
65 {
66     return fields;
67 }
68
69 public String toString()
70 {
71     String r = type + " " + name;
72     if (type.isPrimitive()) r += "=" + value;
73     else if (type.equals(String.class)) r += "=" + value;
74     else if (value == null) r += "=null";
75     return r;
76 }
77 }
```

javax.swing.tree.TreeModel 1.2

- **Object getRoot()**
Возвращает корневой узел.
- **int getChildCount(Object parent)**
Получает количество дочерних узлов заданного родительского узла.
- **Object getChild(Object parent, int index)**
Получает дочерний узел заданного родительского узла по указанному индексу.
- **int getIndexOfChild(Object parent, Object child)**
Получает индекс указанного дочернего узла из заданного родительского узла или значение **-1**, если узел **child** не является дочерним родительского узла **parent** в данной модели дерева.

javax.swing.tree.TreeModel 1.2 (окончание)

- **boolean isLeaf(Object node)**
Возвращает логическое значение `true`, если указанный узел является листом дерева.
- **void addTreeModelListener(TreeModelListener l)**
- **void removeTreeModelListener(TreeModelListener l)**
Вводят или удаляют приемник событий, который уведомляется об изменениях, происходящих в дереве.
- **void valueForPathChanged(TreePath path, Object newValue)**
Вызывается в тех случаях, когда значение узла изменяется в редакторе ячеек дерева.

Параметры:

path

Путь к отредактированному узлу

newValue

Замещающее значение,

возвращаемое редактором ячеек дерева

javax.swing.event.TreeModelListener 1.2

- **void treeNodesChanged(TreeModelEvent e)**
- **void treeNodesInserted(TreeModelEvent e)**
- **void treeNodesRemoved(TreeModelEvent e)**
- **void treeStructureChanged(TreeModelEvent e)**
Вызываются моделью при изменении узлов дерева.

javax.swing.event.TreeModelEvent 1.2

- **TreeModelEvent(Object eventSource, TreePath node)**

Конструирует событие в модели дерева.

Параметры:

eventSource

Модель дерева,

инициирующая данное событие

node

Путь к изменяемому узлу

10.4. Текстовые компоненты

На рис. 10.35 показаны все текстовые компоненты, входящие в состав библиотеки Swing. Три наиболее употребительных компонента из этой библиотеки были рассмотрены ранее: JTextField, JPasswordField и JTextArea (см. главу 9 первого тома настоящего издания). А в этом разделе речь пойдет об остальных текстовых компонентах. Вместе с ними будет рассмотрен компонент JSpinner, содержащий поле отформатированного текста и крошечные кнопки со стрелками вверх и вниз для изменения содержимого этого поля.

Все текстовые компоненты отображают и редактируют данные, хранящиеся в объектной модели класса, реализующего интерфейс Document. Для сохранения последовательности строк простого текста без всякого форматирования в компонентах JTextField и JTextArea применяется класс PlainDocument.

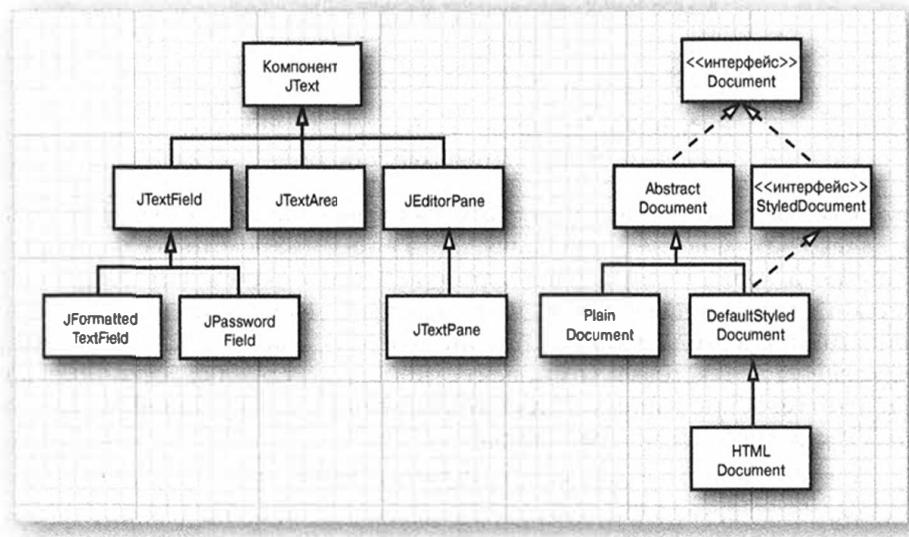


Рис. 10.35. Иерархическая структура текстовых компонентов и документов

Компонент JEditorPane обеспечивает отображение и редактирование стилизованного текста (цветом, шрифтом и пр.) в различных форматах, наиболее распространенным из которых является HTML; см. раздел 10.4.4 далее в этой главе. Интерфейс StyledDocument описывает дополнительные требования к стилям, шрифтам и цветам, а класс HTMLDocument реализует этот интерфейс.

Компонент JTextPane, производный от компонента JEditorPane, также позволяет хранить стилизованный текст и встраиваемые компоненты Swing. Но компонент JTextPane имеет довольно сложную структуру и поэтому здесь не рассматривается. Его подробное описание можно найти в упоминавшейся ранее книге *Core Swing* Кима Топли. А типичный пример применения компонента JTextPane представлен в демонстрационной программе StylePad, входящей в состав JDK.

10.4.1. Отслеживание изменений в текстовых компонентах

Сложности в обращении с интерфейсом Document возникают только при попытке реализовать свой собственный текстовый редактор. Тем не менее именно этот интерфейс чаще всего применяется для отслеживания изменений в текстовых компонентах.

Иногда требуется обновить часть ГПИ, не дожидаясь, пока пользователь щелкнет на кнопке, завершив редактирование текста. Рассмотрим простой пример программы, где отображаются три текстовых поля для красной, синей и зеленой составляющих цвета. Всякий раз, когда изменяется содержимое этих текстовых полей, возникает потребность в немедленном обновлении цвета. На рис. 10.36 показано рабочее окно данной программы, а ее исходный код представлен в листинге 10.19.

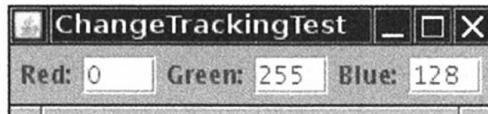


Рис. 10.36. Программа для отслеживания изменений в текстовых полях

Прежде всего следует заметить, что отслеживание нажатий клавиш на клавиатуре — далеко не самая блестящая идея. Дело в том, что некоторые нажатия клавиш (например, клавиш управления курсором) не приводят к изменению текста. Текст может быть обновлен в результате действий мышью (например, средней кнопкой мыши для ввода данных в библиотеке X11). Вместо этого нужно запросить *документ* (а не текстовый компонент), чтобы он уведомлял всякий раз, когда происходит изменение данных. И для этой цели придется установить *приемник событий в документе*, как показано ниже.

```
textField.getDocument().addDocumentListener(listener);
```

После того как текст будет изменен, вызывается один из следующих методов:

```
void insertUpdate(DocumentEvent event)
void removeUpdate(DocumentEvent event)
void changedUpdate(DocumentEvent event)
```

Первые два метода вызываются при вводе или удалении символов. Третий метод вообще не вызывается при редактировании в текстовых полях. Для более сложных видов документов он может быть вызван при ряде других изменений, включая форматирование. К сожалению, уведомить об изменении текста с помощью одиночного обратного вызова нельзя, хотя зачастую и не так важно, каким образом изменяется текст. На этот случай даже не предусмотрен класс адаптера. Таким образом, все три упомянутых выше метода должны быть реализованы в приемнике событий, наступающих в документе. Ниже показано, как это делается в рассматриваемом здесь примере программы. Для получения символьных строк, вводимых пользователем в текстовых полях, а также для установки цвета в методе `setColor()` вызывается метод `getText()`.

```
DocumentListener listener = new DocumentListener()
{
    public void insertUpdate(DocumentEvent event) { setColor(); }
    public void removeUpdate(DocumentEvent event) { setColor(); }
    public void changedUpdate(DocumentEvent event) {}
}
```

Рассматриваемая здесь программа имеет следующее ограничение: пользователь может ввести данные в текстовом поле с ошибками или вообще оставить его пустым. Пока что в данной программе перехватывается исключение типа `NumberFormatException`, генерируемое методом `parseInt()`, и если в текстовом поле введены данные, не являющиеся числовыми, то цвет просто не обновляется. А далее будет показано, как предотвратить ввод пользователем недостоверных данных.



На заметку! Вместо того чтобы принимать события в документе, можно добавить в поле приемник событий действия. Приемнику подобных событий уведомление посыпается всякий раз, когда пользователь нажимает клавишу <Enter>. Но такой прием все же не рекомендуется, поскольку пользователи нередко забывают, что нужно нажать клавишу <Enter> по завершении ввода данных. Если вы все-таки решите воспользоваться приемником событий действия, вам придется также установить приемник фокуса ввода, чтобы не пропустить момент, когда пользователь покинет текстовое поле.

Листинг 10.19. Исходный код из файла `textChange/ColorFrame.java`

```
1 package textChange;
2
3 import java.awt.*;
4 import javax.swing.*;
5 import javax.swing.event.*;
6
7 /**
8  * Фрейм с тремя текстовыми полями для установки цвета фона
9 */
10 public class ColorFrame extends JFrame
11 {
12     private JPanel panel;
13     private JTextField redField;
14     private JTextField greenField;
15     private JTextField blueField;
16
17     public ColorFrame()
18     {
19         DocumentListener listener = new DocumentListener()
20         {
21             public void insertUpdate(DocumentEvent event)
22                 { setColor(); }
23             public void removeUpdate(DocumentEvent event)
24                 { setColor(); }
25             public void changedUpdate(DocumentEvent event) {}}
26         };
27
28     panel = new JPanel();
29
30     panel.add(new JLabel("Red:"));
31     redField = new JTextField("255", 3);
32     panel.add(redField);
33     redField.getDocument().addDocumentListener(listener);
34
35     panel.add(new JLabel("Green:"));
36     greenField = new JTextField("255", 3);
37     panel.add(greenField);
38     greenField.getDocument().addDocumentListener(listener);
39     panel.add(new JLabel("Blue:"));
40     blueField = new JTextField("255", 3);
41     panel.add(blueField);
42     blueField.getDocument().addDocumentListener(listener);
43
44     add(panel);
```

```
44     pack();
45 }
46 /**
47 * Установить цвет фона в соответствии со значениями,
48 * хранящимися в текстовых полях
49 */
50 public void setColor()
51 {
52     try
53     {
54         int red = Integer.parseInt(redField.getText().trim());
55         int green = Integer.parseInt(greenField.getText().trim());
56         int blue = Integer.parseInt(blueField.getText().trim());
57         panel.setBackground(new Color(red, green, blue));
58     }
59     catch (NumberFormatException e)
60     {
61         // не устанавливать цвет, если введенные данные
62         // нельзя проанализировать синтаксически
63     }
64 }
65 }
66 }
```

javax.swing.JComponent 1.2

- Dimension getPreferredSize()
 - void setPreferredSize(Dimension d)

Получают или устанавливают предпочтительные размеры данного компонента.

javax.swing.text.Document 1.2

- **int getLength()**
Возвращает количество символов, которые документ содержит в данный момент.
 - **String getText(int offset, int length)**
Возвращает текст, содержащийся в данной части документа.
Параметры: **offset** Начало текста
 length Требуемая длина символьной строки
 - **void addDocumentListener(DocumentListener listener)**
Регистрирует приемник событий для получения уведомлений об изменениях в документе.

javax.swing.event.DocumentEvent 1.2

- **Document getDocument()**
Получает документ, являющийся источником событий.

javax.swing.event.DocumentListener 1.2

- **void changedUpdate(DocumentEvent event)**
Вызывается всякий раз, когда изменяется атрибут или ряд атрибутов.
- **void insertUpdate(DocumentEvent event)**
Вызывается всякий раз, когда текст вводится в документ.
- **void removeUpdate(DocumentEvent event)**
Вызывается всякий раз, когда удаляется часть документа.

10.4.2. Поля ввода форматируемого текста

В предыдущем примере программы требовалось, чтобы пользователь вводил числа, а не произвольные символьные строки, т.е. ему разрешалось вводить только цифры от 0 до 9 и дефис (-). Но если дефис вообще присутствует, то он должен быть *первым* символом во вводимой строке.

На первый взгляд, проверка такого ввода не является сложной задачей. Так, в текстовом поле можно установить приемник событий от клавиш, а затем удалить все события от клавиш, которые не связаны с вводом цифр или дефиса. К сожалению, этот простой способ, хоть он обычно и рекомендуется для проверки вводимых данных, на практике оказывается ненадежным. Прежде всего, не всякая комбинация допустимых для ввода символов является достоверным числом. Например, комбинации символов --3 и 3-3 не являются достоверными комбинациями, несмотря на то, что они состоят из допустимых символов. Но самое главное, что имеются и другие способы изменения текста, не подразумевающие ввод символов с клавиатуры. В зависимости от предпочтений пользователя для вырезания, копирования и вставки текста могут использоваться определенные комбинации клавиш. Например, в визуальном стиле Metal с помощью комбинации клавиш <Ctrl+V> содержимое из буфера обмена вставляется в текстовое поле. Это означает, что нужно следить и за тем, чтобы пользователь не вставил недостоверный символ. Ясно, что попытка отфильтровывать нажатия клавиш с целью гарантировать достоверность содержимого текстового поля может превратиться в настоящий кошмар. Поэтому разработчики прикладных программ должны быть избавлены от подобных хлопот.

Как ни странно, но до выхода версии Java SE 1.4 не существовало компонентов, предназначенных для ввода числовых значений. В первом издании данной книги мы предоставили реализацию текстового поля типа `IntTextField` для ввода целого числа с подходящим форматированием. В каждом последующем издании мы изменяли эту реализацию, чтобы воспользоваться любым преимуществом от еще незрелых схем проверки, внедрявшихся в каждую версию Java. И наконец, в версии Java SE 1.4 разработчики библиотеки Swing внедрили класс `JFormattedTextField`, который можно использовать для ввода не только чисел, но и дат или еще более экзотических форматируемых значений вроде IP-адресов.

10.4.2.1. Ввод целочисленных значений

Начнем с простого случая, когда текстовое поле используется для ввода целочисленных значений следующим образом:

```
JFormattedTextField intField =  
    new JFormattedTextField(NumberFormat.getIntegerInstance());
```

Метод `NumberFormat.getIntegerInstance()` возвращает объект, который форматирует целые числа с учетом текущих региональных настроек. В США, например, запятые используются в качестве десятичных разделителей, позволяя вводить значения вроде 1,729. В главе 7 уже пояснялось, каким образом можно сменить текущие региональные настройки.

Как и для любого другого текстового поля, в данном случае можно установить несколько столбцов. Ниже показано, как это делается.

```
intField.setColumns(6);
```

Значение по умолчанию можно задать с помощью метода `setValue()`. Этот метод принимает параметр типа `Object`, поэтому значение типа `int` необходимо представить как объект типа `Integer` следующим образом:

```
intField.setValue(new Integer(100));
```

Как правило, пользователи вводят данные в нескольких текстовых полях, а затем щелкают на кнопке, чтобы все введенные значения были прочитаны. После активизации кнопки щелчком на ней можно получить введенное пользователем значение, вызвав метод `getValue()`. Этот метод возвращает результат типа `Object`, поэтому его придется привести к соответствующему типу. Из текстового поля типа `JFormattedTextField` возвращается объект типа `Long`, если пользователь отредактировал значение. А если пользователь не вносил никаких изменений, то возвращается исходный объект типа `Integer`. Таким образом, возвращаемое значение необходимо привести к обычному суперклассу `Number`, как показано ниже.

```
Number value = (Number) intField.getValue();  
int v = value.intValue();
```

Поле ввода форматируемого текста не представляет особого интереса до тех пор, пока не возникнет потребность принять во внимание то, что может произойти, если пользователь введет недостоверные данные. Этот вопрос обсуждается в следующем подразделе.

10.4.2.2. Поведение при потере фокуса ввода

Попробуем выяснить, что же происходит, когда пользователь вводит данные в текстовом поле. Если пользователь введет какие-нибудь данные и в какой-то момент решит покинуть текстовое поле, возможно, щелкнув кнопкой мыши на другом компоненте, то текстовое поле *потеряет фокус ввода*. Мигающий курсор больше не появляется в текстовом поле, а нажатия клавиш на клавиатуре направляются другому компоненту.

Когда поле ввода форматируемого текста теряет фокус ввода, средство формирования анализирует текстовую строку, введенную пользователем. Если этому средству известно, как преобразовать текстовую строку в объект, то введенный

текст считается достоверным, а иначе — недостоверным. Чтобы проверить, является ли текущее содержимое текстового поля достоверным, можно вызвать метод `isValid()`.

Поведение по умолчанию при потере фокуса называется “принять или возвратить”. Если текстовая строка оказывается достоверной, она *принимается*. Средство форматирования преобразует ее в объект, который становится текущим значением поля (т.е. значением, возвращаемым из метода `getValue()`, как было показано в предыдущем подразделе). Затем это значение преобразуется обратно в символьную строку, которая становится текстовой строкой, отображаемой в поле ввода. Например, средство форматирования целочисленных значений распознает введенную комбинацию **1729** как достоверную, устанавливает текущее значение в виде объекта `new Long(1729)`, а затем преобразует его обратно в символьную строку с запятой в качестве десятичного разделителя: **1, 729**.

А если текстовая строка оказывается недостоверной, то текущее значение не меняется и текстовое поле *возвращаетется* к символьной строке, представляющей прежнее значение. Так, если пользователь введет неверное значение **вроде x1**, при потере фокуса ввода будет восстановлено прежнее значение.



На заметку! Средство форматирования целых чисел считает текстовую строку достоверной, если она начинается с целого числа. Например, строка **1729x** является действительной. Она преобразуется в число **1729**, которое затем преобразуется в символьную строку **1, 729**.

С помощью метода `setFocusLostBehavior()` можно установить другие виды поведения. Поведение “принять” немногим отличается от поведения по умолчанию. Если текстовая строка оказывается недостоверной, то сама текстовая строка и соответствующее ей значение в текстовом поле остаются без изменения, но с этого момента они перестают быть синхронизированными. Поведение “продолжить” является еще более консервативным. Даже если текстовая строка оказывается недостоверной, ни текстовое поле, ни текущее значение не изменяется. Чтобы синхронизировать их снова, можно даже вызвать метод `commitEdit()` или `setText()`. И наконец, имеется еще и поведение “возвратить”, хотя пользы от него немного. Всякий раз, когда теряется фокус ввода, данные, введенные пользователем, игнорируются, а текстовая строка возвращается к старому значению.



На заметку! Как правило, по умолчанию используется поведение “принять или возвратить”, но оно чревато серьезными осложнениями. Допустим, что диалоговое окно содержит текстовое поле для ввода целочисленных значений. Пользователь вводит текстовую строку **“1729”** с пробелом в качестве первого символа и щелкает на кнопке **OK**. Этот пробел делает введенное числовое значение недостоверным, и поэтому текстовое поле возвращается к старому значению. Приемник событий от кнопки **OK** получает значение из текстового поля и просто закрывает диалоговое окно, а пользователь так и не узнает, что новое значение было отклонено. В подобных случаях следует выбрать поведение “принять” и принудить приемник событий от кнопки **OK** проверить все правки в текстовом поле на достоверность, прежде чем закрывать диалоговое окно.

10.4.2.3. Фильтры

Основные функции полей ввода форматируемого текста просты и достаточноны для большинства случаев их применения. Тем не менее их можно дополнить

некоторыми усовершенствованиями. Допустим, требуется сделать так, чтобы пользователь вообще не смог ввести нецифровые значения. Этого можно добиться с помощью *фильтра документа*. Напомним, что контроллер в проектном шаблоне “модель–представление–контроллер” преобразует события ввода в команды и изменяет базовый документ в текстовом поле, т.е. текстовую строку, хранящуюся в объекте типа PlainDocument. Например, всякий раз, когда контроллер обрабатывает команду вставки текста в документ, он вызывает команду “вставить строку”. Стока, которую требуется вставить, может быть представлена одним символом или содержимым буфера вставки. Фильтр документа может перехватить эту команду и изменить символьную строку или вообще отменить ее вставку. Ниже приведен исходный код метода insertString() для фильтра, анализирующего вставляемую символьную строку и вставляющего только те символы, которые являются цифрами или знаком -. (В этом методе обрабатываются дополнительные символы в Юникоде, как пояснялось в главе 3 первого тома настоящего издания. А описание класса StringBuilder приведено в главе 2 настоящего тома.)

```
public void insertString(FilterBypass fb, int offset, String string,
    AttributeSet attr) throws BadLocationException
{
    StringBuilder builder = new StringBuilder(string);
    for (int i = builder.length() - 1; i >= 0; i--)
    {
        int cp = builder.codePointAt(i);
        if (!Character.isDigit(cp) && cp != '-')
        {
            builder.deleteCharAt(i);
            if (Character.isSupplementaryCodePoint(cp))
            {
                i--;
                builder.deleteCharAt(i);
            }
        }
    }
    super.insertString(fb, offset, builder.toString(), attr);
}
```

Необходимо также переопределить метод replace() из класса DocumentFilter. Он вызывается при выборе текста и последующей его вставке. Реализация метода replace() не представляет особых трудностей, как будет показано далее, в листинге 10.21.

Затем следует установить фильтр документа. К сожалению, простого способа сделать это не существует. Для этого придется сначала переопределить метод getDocumentFilter() из класса, реализующего средство форматирования, а затем передать форматирующий объект этого класса в текстовое поле типа JFormattedTextField. Так, в текстовом поле ввода целочисленных значений используется средство форматирования типа InternationalFormatter, которое инициализируется с помощью метода NumberFormat.getIntegerInstance(). В приведенном ниже фрагменте кода показано, каким образом осуществляется установка средства форматирования для получения требуемого фильтра.

```
JFormattedTextField intField = new JFormattedTextField(
    new InternationalFormatter(NumberFormat.getIntegerInstance()))
```

```
{  
    private DocumentFilter filter = new IntFilter();  
    protected DocumentFilter getDocumentFilter()  
    {  
        return filter;  
    }  
});
```



На заметку! В документации на Java SE сказано, что класс `DocumentFilter` разработан с целью избавиться от лишней подклассификации. До версии Java SE 1.3 фильтрация в текстовом поле достигалась путем расширения класса `PlainDocument` и переопределения методов `insertString()` и `replace()`. Теперь же в классе `PlainDocument` имеется подключаемый фильтр. И это, конечно, замечательное усовершенствование, но оно могло бы быть еще лучше, если бы фильтр был сделан подключаемым и в классе, реализующем средство форматирования. Увы, этого не было сделано, и поэтому приходится выполнять подклассификацию класса, реализующего средство форматирования.

Попробуйте выполнить пример программы `FormatTest`, исходный код которой представлен в листинге 10.20. В третьем текстовом поле этой программы установлен фильтр. В этом поле допускается вставлять только цифры или знак "минус" (-), но по-прежнему можно вводить недостоверные символьные строки вроде "1-2-3". Вообще говоря, фильтрация не позволяет застраховаться от ввода всех недостоверных символьных строк. Например, строка "-" является недостоверной, хотя фильтр не может отклонить ее, поскольку это приставка к достоверной строке "-1". И хотя фильтры не в состоянии обеспечить идеальную защиту от ввода недостоверных данных, пользоваться ими все же следует для отклонения тех вводимых данных, которые явно недействительны.



Совет! Фильтрация применяется и для того, чтобы перевести все символы в верхний регистр. Написать код для такого фильтра нетрудно. В методах `insertString()` и `replace()` из класса этого фильтра необходимо перевести символы вставляемой строки в верхний регистр, а затем вызвать соответствующий метод из суперкласса.

10.4.2.4. Средства проверки достоверности

Существует еще один потенциально полезный механизм уведомления пользователей о неверно введенных данных. К любому компоненту типа `JComponent` можно присоединить *средство проверки достоверности*. Если компонент теряет фокус ввода, происходит обращение к средству проверки достоверности. И если это средство уведомляет о недостоверности содержимого компонента, то фокус ввода сразу же возвращается к данному компоненту. Таким образом, пользователю придется устраниТЬ ошибку во введенных данных, прежде чем вводить другие данные.

Средство проверки достоверности должно расширять абстрактный класс `InputVerifier` и определять метод `verify()`. Определить средство проверки достоверности для полей ввода форматируемого текста совсем не трудно. Метод `isValid()` из класса `JFormattedTextField` вызывает средство форматирования и возвращает логическое значение `true`, если это средство преобразует текстовую строку в объект. В приведенном ниже фрагменте кода показано, каким образом средство проверки достоверности присоединяется к текстовому полю типа `JFormattedTextField`.

```

intField.setInputVerifier(new InputVerifier()
{
    public boolean verify(JComponent component)
    {
        JFormattedTextField field = (JFormattedTextField) component;
        return field.isEditValid();
    }
});

```

Четвертое текстовое поле в рассматриваемом здесь примере программы имеет средство проверки достоверности. Попробуйте ввести недействительное число (например, **x1729**) и нажмите клавишу **<Tab>** или щелкните кнопкой мыши на другом текстовом поле. Обратите внимание на то, что поле тут же получит фокус ввода. Но если вы щелкнете на кнопке **OK**, приемник действий вызовет метод **getValue()**, который сообщит о последнем действительном значении.

Но и средство проверки достоверности нельзя считать совершенно надежным. Если щелкнуть на кнопке, она уведомит приемники действий до того, как компонент с недействительными данными снова получит фокус ввода. И тогда приемники действий могут получить недействительный результат из компонента, который не прошел проверку на достоверность. Подобное поведение объясняется тем, что пользователь может при желании щелкнуть на кнопке **Cancel** (Отмена), так и не устранив ошибку во введенных данных.

10.4.2.5. Другие стандартные средства форматирования

Помимо средств форматирования целочисленных значений, в классе **JFormattedTextField** поддерживается также ряд других средств форматирования числовых значений с плавающей точкой, в процентах и денежных единицах. Для этой цели в классе **NumberFormat** имеются следующие статические методы:

```

getNumberInstance()
getCurrencyInstance()
getPercentInstance()

```

Например, чтобы получить текстовое поле для ввода денежных сумм, достаточно вызвать следующий метод:

```

JFormattedTextField currencyField =
    new JFormattedTextField(NumberFormat.getCurrencyInstance());

```

А для редактирования даты и времени достаточно вызвать один из следующих статических методов из класса **DateFormat**:

```

getDateInstance()
getTimeInstance()
getDateTimeInstance()

```

Например, в приведенном ниже поле дата редактируется в принятом по умолчанию так называемом “среднем” формате **Aug 5, 2007**.

```

JFormattedTextField dateField =
    new JFormattedTextField(DateFormat.getDateInstance());

```

Вместо него можно выбрать другой, так называемый “короткий” формат **8/5/07**, вызвав следующий метод:

```

DateFormat.getDateInstance(DateFormat.SHORT)

```

 **На заметку!** По умолчанию для дат выбирается так называемый "нестрогий" формат. Это означает, что несуществующая дата вроде **February 31, 2002** будет приведена к дате **March 3, 2002**. Подобное поведение может показаться необычным для пользователей. В таком случае следует вызвать метод `setLenient(false)` для объекта типа `DateFormat`.

Стандартное средство форматирования типа `DefaultFormatter` позволяет форматировать объекты любого класса, имеющего конструктор со строковым параметром и соответствующий метод `toString()`. Например, в классе `URL` имеется конструктор `URL(String)`, который можно использовать для составления `URL` из символьной строки следующим образом:

```
URL url = new URL("http://horstmann.com");
```

Таким образом, стандартное средство типа `DefaultFormatter` можно использовать для форматирования объектов типа `URL`. Это средство вызывает метод `toString()` со значением поля, чтобы инициализировать текст в данном поле. Когда поле теряет фокус ввода, средство форматирования создает новый объект того же класса, что и класс текущего значения, используя конструктор с параметром типа `String`. Если этот конструктор генерирует исключение, результат редактирования в данном поле считается недостоверным. Можете сами убедиться в этом, введя `URL` без префикса "`http:`" в рассматриваемом здесь примере программы.

 **На заметку!** По умолчанию стандартное средство форматирования типа `DefaultFormatter` работает в режиме перезаписи. Этим оно отличается от остальных средств форматирования. Но пользы от такого режима немного, поэтому его лучше отключить, вызвав метод `setOverwriteMode(false)`.

Наконец, средство форматирования типа `MaskFormatter` оказывается удобным для форматирования по шаблонам фиксированного размера, состоящим из констант и переменных символов. Например, номера карточек социального страхования в США (вроде **078-05-1120**) могут быть отформатированы по приведенному ниже шаблону, где знак `#` обозначает одну цифру. В табл. 10.3 перечислены символы, которые можно использовать в средстве форматирования по шаблону.

```
new MaskFormatter("###-##-####")
```

Таблица 10.3. Символы, употребляемые в средстве форматирования типа `MaskFormatter`

Символ	Описание
<code>#</code>	Цифра
<code>?</code>	Буква
<code>U</code>	Буква, преобразуемая в прописную
<code>L</code>	Буква, преобразуемая в строчную
<code>A</code>	Буква или цифра
<code>H</code>	Шестнадцатеричная цифра [0-9A-Fa-f]
<code>*</code>	Любой символ
<code>'</code>	Символ перехода, позволяющий включить специальный символ в шаблон, экранируя его

На символы, которые допускается вводить в поле, можно наложить определенные ограничения, вызвав один из следующих методов из класса `MaskFormatter`:

```
setValidCharacters()  
setInvalidCharacters()
```

Так, чтобы прочитать буквенную оценку академической успеваемости (например, A+ или F), можно было бы воспользоваться приведенным ниже фрагментом кода. Но указать, что второй символ не может быть буквой, никак нельзя.

```
MaskFormatter formatter = new MaskFormatter("U*");  
formatter.setValidCharacters("ABCDF+- ");
```

Следует, однако, иметь в виду, что символьная строка, форматируемая средством форматирования по шаблону, имеет такую же длину, как и у шаблона. Если пользователь удаляет символы во время редактирования, они будут заменяться *символом-заполнителем*. По умолчанию в качестве такого символа используется пробел, но его можно заменить другим символом, вызвав метод `setPlaceholderCharacter()`, как показано ниже.

```
formatter.setPlaceholderCharacter('0');
```

По умолчанию средство форматирования по шаблону работает в режиме замены вводимых данных, понять который можно интуитивно, если опробовать его в рассматриваемом здесь примере программы. Следует также иметь в виду, что позиция вставки переходит через фиксированные символы в шаблоне.

Средства форматирования по шаблону оказываются довольно эффективными, когда применяются строгие шаблоны вроде номеров карточек социального страхования и номеров телефонов в США. Но, поскольку в шаблонах не допускается никаких отклонений, средство форматирования по шаблону не годится, например, для международных номеров телефонов, имеющих разное количество цифр.

10.4.2.6. Специальные средства форматирования

Если ни одно из стандартных средств форматирования не подходит, можно без труда определить собственное. Рассмотрим, например, 4-байтовые IP-адреса вроде следующего:

130.65.86.66

Для форматирования подобных адресов по шаблону нельзя воспользоваться классом `MaskFormatter`, поскольку каждый байт в IP-адресе может быть представлен одной, двумя или тремя цифрами. Кроме того, в средстве форматирования необходимо проверить, не превышает ли значение в каждом байте величину 255. Чтобы определить собственное средство форматирования, нужно расширить класс `DefaultFormatter` и переопределить следующие методы:

```
String valueToString(Object value)  
Object stringToValue(String text)
```

Первый метод преобразует значение из текстового поля в символьную строку, которая затем отображается в этом поле. А второй метод анализирует введенный пользователем текст и преобразует его обратно в объект. Если в одном из этих методов обнаружится ошибка, должно быть сгенерировано исключение типа `ParseException`.

В рассматриваемом здесь примере программы IP-адреса сохраняются в массиве `byte[]` длиной, равной 4. Метод `valueToString()` формирует символьную строку, разделяя в ней байты точками. Однако значения типа `byte` могут находиться в пределах от -128 до 127. (Например, в IP-адресе 130.65.86.66 первый октет на самом деле является байтом со значением -126.) Чтобы преобразовать отрицательные значения байтов в целочисленные значения без знака, к ним следует прибавить значение 256:

```
public String valueToString(Object value) throws ParseException
{
    if (!(value instanceof byte[]))
        throw new ParseException("Not a byte[]", 0);
    byte[] a = (byte[]) value;
    if (a.length != 4)
        throw new ParseException("Length != 4", 0);
    StringBuilder builder = new StringBuilder();
    for (int i = 0; i < 4; i++)
    {
        int b = a[i];
        if (b < 0) b += 256;
        builder.append(String.valueOf(b));
        if (i < 3) builder.append('.');
    }
    return builder.toString();
}
```

С другой стороны, метод `stringToValue()` осуществляет синтаксический анализ символьной строки и выдает объект типа `byte[]`, если эта строка оказывается достоверной. В противном случае генерируется исключение типа `ParseException`, как показано в приведенном ниже фрагменте кода.

```
public Object stringToValue(String text) throws ParseException
{
    StringTokenizer tokenizer = new StringTokenizer(text, ".");
    byte[] a = new byte[4];
    for (int i = 0; i < 4; i++)
    {
        int b = 0;
        try
        {
            b = Integer.parseInt(tokenizer.nextToken());
        }
        catch (NumberFormatException e)
        {
            throw new ParseException("Not an integer", 0);
        }
        if (b < 0 || b >= 256)
            throw new ParseException("Byte out of range", 0);
        a[i] = (byte) b;
    }
    return a;
}
```

Попробуйте ввести данные в поле IP-адреса из рассматриваемого здесь примера программы. Если вы введете недостоверный адрес, в этом поле отобразится последний достоверный адрес. Весь исходный код специального средства форматирования представлен в листинге 10.22, а в примере программы из листинга

10.20 демонстрируется применение различных полей ввода форматируемого текста (рис. 10.37). Чтобы извлечь текущие значения из этих полей, достаточно щелкнуть на кнопке ОК.



На заметку! По адресу www.oracle.com/technetwork/java/reftf-138955.html доступна краткая статья, где описывается средство форматирования, для которого подходит любое регулярное выражение.

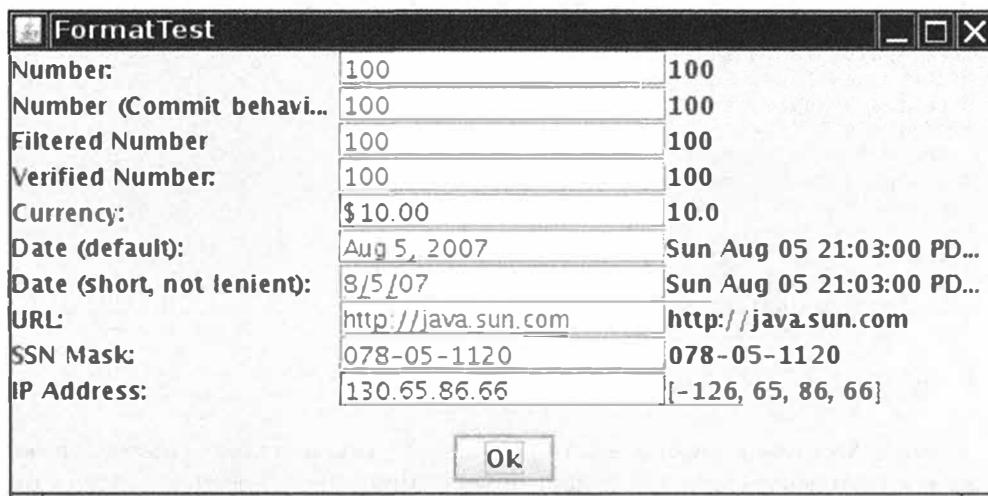


Рис. 10.37. Программа FormatTest

Листинг 10.20. Исходный код из файла textFormat/FormatTestFrame.java

```

1 package textFormat;
2
3 import java.awt.*;
4 import java.net.*;
5 import java.text.*;
6 import java.util.*;
7 import javax.swing.*;
8 import javax.swing.text.*;
9
10 /**
11 * Фрейм с целым рядом полей ввода форматируемого текста и
12 * кнопкой для отображения значений в этих полях
13 */
14 public class FormatTestFrame extends JFrame
15 {
16     private DocumentFilter filter = new IntFilter();
17     private JButton okButton;
18     private JPanel mainPanel;
19
20     public FormatTestFrame()
21     {
22         JPanel buttonPanel = new JPanel();
23         okButton = new JButton("Ok");

```

```
24     buttonPanel.add(okButton);
25     add(buttonPanel, BorderLayout.SOUTH);
26
27     mainPanel = new JPanel();
28     mainPanel.setLayout(new GridLayout(0, 3));
29     add(mainPanel, BorderLayout.CENTER);
30
31     JFormattedTextField intField = new JFormattedTextField(
32         NumberFormat.getIntegerInstance());
33     intField.setValue(new Integer(100));
34     addRow("Number:", intField);
35
36     JFormattedTextField intField2 = new JFormattedTextField(
37         NumberFormat.getIntegerInstance());
38     intField2.setValue(new Integer(100));
39     intField2.setFocusLostBehavior(JFormattedTextField.COMMIT);
40     addRow("Number (Commit behavior):", intField2);
41
42     JFormattedTextField intField3 = new JFormattedTextField(
43         new InternationalFormatter(
44             NumberFormat.getIntegerInstance()))
45     {
46         protected DocumentFilter getDocumentFilter()
47         {
48             return filter;
49         }
50     });
51     intField3.setValue(new Integer(100));
52     addRow("Filtered Number", intField3);
53
54     JFormattedTextField intField4 = new JFormattedTextField(
55         NumberFormat.getIntegerInstance());
56     intField4.setValue(new Integer(100));
57     intField4.setInputVerifier(new InputVerifier()
58     {
59         public boolean verify(JComponent component)
60         {
61             JFormattedTextField field =
62                 (JFormattedTextField) component;
63             return field.isEditValid();
64         }
65     });
66     addRow("Verified Number:", intField4);
67
68     JFormattedTextField currencyField = new JFormattedTextField(
69         NumberFormat.getCurrencyInstance());
70     currencyField.setValue(new Double(10));
71     addRow("Currency:", currencyField);
72
73     JFormattedTextField dateField = new JFormattedTextField(
74         DateFormat.getDateInstance());
75     dateField.setValue(new Date());
76     addRow("Date (default):", dateField);
77
78     DateFormat format = DateFormat.getDateInstance(
79         DateFormat.SHORT);
80     format.setLenient(false);
81     JFormattedTextField dateField2 =
```

```
82             new JFormattedTextField(format);
83     dateField2.setValue(new Date());
84     addRow("Date (short, not lenient):", dateField2);
85
86     try
87     {
88         DefaultFormatter formatter = new DefaultFormatter();
89         formatter.setOverwriteMode(false);
90         JFormattedTextField urlField =
91             new JFormattedTextField(formatter);
92         urlField.setValue(new URL("http://java.sun.com"));
93         addRow("URL:", urlField);
94     }
95     catch (MalformedURLException ex)
96     {
97         ex.printStackTrace();
98     }
99
100    try
101    {
102        MaskFormatter formatter = new MaskFormatter("###-##-####");
103        formatter.setPlaceholderCharacter('0');
104        JFormattedTextField ssnField =
105            new JFormattedTextField(formatter);
106        ssnField.setValue("078-05-1120");
107        addRow("SSN Mask:", ssnField);
108    }
109    catch (ParseException ex)
110    {
111        ex.printStackTrace();
112    }
113
114    JFormattedTextField ipField = new JFormattedTextField(
115        new IPAddressFormatter());
116    ipField.setValue(new byte[] { (byte) 130, 65, 86, 66 });
117    addRow("IP Address:", ipField);
118    pack();
119 }
120
121 /**
122 * Вводит ряд полей на основной панели
123 * @param labelText Метка поля
124 * @param field Образцовое поле
125 */
126 public void addRow(String labelText,
127                     final JFormattedTextField field)
128 {
129     mainPanel.add(new JLabel(labelText));
130     mainPanel.add(field);
131     final JLabel valueLabel = new JLabel();
132     mainPanel.add(valueLabel);
133     okButton.addActionListener(new ActionListener()
134     {
135         public void actionPerformed(ActionEvent event)
136         {
137             Object value = field.getValue();
138             Class<?> cl = value.getClass();
139             String text = null;
```

```

140         if (cl.isArray())
141         {
142             if (cl.getComponentType().isPrimitive())
143             {
144                 try
145                 {
146                     text = Arrays.class.getMethod("toString", cl)
147                         .invoke(null, value).toString();
148                 }
149                 catch (ReflectiveOperationException ex)
150                 {
151                     // игнорировать ожидаемые результаты рефлексии
152                 }
153             }
154             else text = Arrays.toString((Object[]) value);
155         }
156         else text = value.toString();
157         valueLabel.setText(text);
158     }
159 });
160 }
161 }
```

Листинг 10.21. Исходный код из файла `textFormat/IntFilter.java`

```

1 package textFormat;
2
3 import javax.swing.text.*;
4
5 /**
6  * Фильтр, ограничивающий ввод данных цифрами и знаком '-'
7  */
8 public class IntFilter extends DocumentFilter
9 {
10    public void insertString(FilterBypass fb, int offset,
11        String string, AttributeSet attr) throws BadLocationException
12    {
13        StringBuilder builder = new StringBuilder(string);
14        for (int i = builder.length() - 1; i >= 0; i--)
15        {
16            int cp = builder.codePointAt(i);
17            if (!Character.isDigit(cp) && cp != '-')
18            {
19                builder.deleteCharAt(i);
20                if (Character.isSupplementaryCodePoint(cp))
21                {
22                    i--;
23                    builder.deleteCharAt(i);
24                }
25            }
26        }
27        super.insertString(fb, offset, builder.toString(), attr);
28    }
29
30    public void replace(FilterBypass fb, int offset, int length,
31        String string, AttributeSet attr) throws BadLocationException
32    {
```

```

33     if (string != null)
34     {
35         StringBuilder builder = new StringBuilder(string);
36         for (int i = builder.length() - 1; i >= 0; i--)
37         {
38             int cp = builder.codePointAt(i);
39             if (!Character.isDigit(cp) && cp != '-')
40             {
41                 builder.deleteCharAt(i);
42                 if (Character.isSupplementaryCodePoint(cp))
43                 {
44                     i--;
45                     builder.deleteCharAt(i);
46                 }
47             }
48         }
49         string = builder.toString();
50     }
51     super.replace(fb, offset, length, string, attr);
52 }
53 }
```

Листинг 10.22. Исходный код из файла `textFormat/IPAddressFormatter.java`

```

1 package textFormat;
2
3 import java.text.*;
4 import java.util.*;
5 import javax.swing.text.*;
6
7 /**
8  * Средство форматирования 4-байтовых IP-адресов в форме a.b.c.d
9  */
10 public class IPAddressFormatter extends DefaultFormatter
11 {
12     public String valueToString(Object value) throws ParseException
13     {
14         if (!(value instanceof byte[]))
15             throw new ParseException("Not a byte[]", 0);
16         byte[] a = (byte[]) value;
17         if (a.length != 4) throw new ParseException("Length != 4", 0);
18         StringBuilder builder = new StringBuilder();
19         for (int i = 0; i < 4; i++)
20         {
21             int b = a[i];
22             if (b < 0) b += 256;
23             builder.append(String.valueOf(b));
24             if (i < 3) builder.append('.');
25         }
26         return builder.toString();
27     }
28
29     public Object stringToValue(String text) throws ParseException
30     {
31         StringTokenizer tokenizer = new StringTokenizer(text, ".");
32         byte[] a = new byte[4];
33         for (int i = 0; i < 4; i++)
```

```
34  {
35      int b = 0;
36      if (!tokenizer.hasMoreTokens())
37          throw new ParseException("Too few bytes", 0);
38      try
39      {
40          b = Integer.parseInt(tokenizer.nextToken());
41      }
42      catch (NumberFormatException e)
43      {
44          throw new ParseException("Not an integer", 0);
45      }
46      if (b < 0 || b >= 256) throw new ParseException(
47          "Byte out of range", 0);
48      a[i] = (byte) b;
49  }
50  if (tokenizer.hasMoreTokens()) throw new ParseException(
51      "Too many bytes", 0);
52 return a;
53 }
54 }
```

`javax.swing.JFormattedTextField 1.4`

- **`JFormattedTextField(Format fmt)`**
- Создает текстовое поле, в котором используется указанный формат.
- **`JFormattedTextField(JFormattedTextField.AbstractFormatter formatter)`**
Создает текстовое поле, в котором используется указанное средство форматирования. Следует, однако, иметь в виду, что классы `DefaultFormatter` и `InternationalFormatter` являются производными от класса `JFormattedTextField.AbstractFormatter`.
- **`Object getValue()`**
Возвращает текущее достоверное значение из поля. Однако оно может не соответствовать символьной строке, редактируемой в данном поле.
- **`void setValue(Object value)`**
Пытается установить значение для указанного объекта. Попытка окажется неудачной, если средству форматирования не удастся преобразовать объект в символьную строку.
- **`void commitEdit()`**
Пытается установить достоверное значение символьной строки, редактируемой в поле. Попытка может оказаться неудачной, если средству форматирования не удастся преобразовать символьную строку.
- **`boolean isEditValid()`**
Проверяет, представляет ли редактируемая символьная строка достоверное значение.
- **`int getFocusLostBehavior()`**
- **`void setFocusLostBehavior(int behavior)`**
Получают или устанавливают поведение "потеря фокуса ввода". Параметр `behavior` может принимать значения следующих констант из класса `JFormattedTextField`: `COMMIT_OR_REVERT`, `REVERT`, `COMMIT` и `PERSIST`.

javax.swing.JFormattedTextField.AbstractFormatter 1.4

- **abstract String valueToString(Object value)**
Преобразует значение в редактируемую символьную строку. Генерирует исключение типа `ParseException`, если значение не подходит для данного средства форматирования.
- **abstract Object stringToValue(String s)**
Преобразует символьную строку в значение. Генерирует исключение типа `ParseException`, если формат строки *s* неподходящий.
- **DocumentFilter getDocumentFilter()**
Этот метод следует переопределить, чтобы создать фильтр документа, ограничивающий ввод данных в текстовом поле. Возвращаемое пустое значение `null` указывает на то, что фильтрация не нужна.

javax.swing.text.DefaultFormatter 1.3

- **boolean getOverwriteMode()**
- **void setOverwriteMode(boolean mode)**
Получают или устанавливают режим перезаписи. Если параметр *mode* принимает логическое значение `true`, новые символы перезапишут прежние символы при редактировании текста.

javax.swing.text.DocumentFilter 1.4

- **void insertString(DocumentFilter.FilterBypass bypass, int offset, String text, AttributeSet attrib)**
Вызывается перед вставкой символьной строки в документ. Этот метод можно переопределить и видоизменить символьную строку. Вставку можно отменить, не вызывая метод `super.insertString()` или же вызывая методы объекта *bypass*, чтобы видоизменить документ без фильтрации.
Параметры: **bypass** Объект, позволяющий выполнять команды редактирования в обход фильтра
offset Смещение, с которого начинается вставка текста
text Вставляемые символы
attrib Атрибуты форматирования вставляемого текста
- **void replace(DocumentFilter.FilterBypass bypass, int offset, int length, String text, AttributeSet attrib)**
Вызывается перед заменой части документа новой символьной строкой. Этот метод можно переопределить и видоизменить символьную строку. Замену можно отменить, не вызывая метод `super.replace()` или же вызывая объект *bypass*, чтобы видоизменить документ без фильтрации.

- | | | |
|------------|---------------|--|
| Параметры: | bypass | Объект, позволяющий выполнять команды редактирования в обход фильтра |
| | offset | Смещение, с которого начинается замена части документа |

javax.swing.text.DocumentFilter 1.4 (окончание)

length	Длина заменяемой части документа
text	Заменяющие символы
attrib	Атрибуты форматирования заменяющего текста
• void remove (DocumentFilter.FilterBypass bypass, int offset, int length)	
	Вызывается перед удалением части документа. Если требуется проанализировать результат удаления, документ можно получить, вызвав метод <code>bypass.getDocument()</code> .
Параметры:	
bypass	Объект, позволяющий выполнять команды редактирования в обход фильтра
offset	Смещение, с которого начинается удаление части документа
length	Длина удаляемой части документа

javax.swing.text.MaskFormatter 1.4**• MaskFormatter(String mask)**

Создает средство форматирования по заданному шаблону. Символы, которые допускается использовать в шаблоне, перечислены в табл. 10.3.

• String getValidCharacters()**• void setValidCharacters(String characters)**

Получают или устанавливают символы, допустимые для редактирования. Для переменных частей шаблона допускается использовать только символы из указанной строки.

• String getInvalidCharacters()**• void setInvalidCharacters(String characters)**

Получают или устанавливают символ-заполнитель, используемый для тех переменных символов в шаблоне, которые пользователь еще не ввел. По умолчанию символом-заполнителем служит пробел.

• char getPlaceholderCharacter()**• void setPlaceholderCharacter(char ch)**

Получают или устанавливают строку-заполнитель. Конечная часть этой строки используется в том случае, если пользователь не ввел все переменные символы в шаблоне. Если эта строка имеет пустое значение `null` или короче шаблона, остальная часть вводимых данных заполняется символом-заполнителем.

• boolean getValueContainsLiteralCharacters()**• void setValueContainsLiteralCharacters(boolean b)**

Получают или устанавливают флаг "значение содержит буквенные символы". Если этот флаг принимает логическое значение `true`, то значение из поля содержит буквенные {неизменяемые} части шаблона. А если этот флаг принимает логическое значение `false`, то буквенные символы удаляются. По умолчанию флаг принимает логическое значение `true`.

10.4.3. Компонент JSpinner

Компонент JSpinner содержит счетчик в виде текстового поля и две небольшие кнопки со стрелками справа от него. Если щелкнуть на этих кнопках, значение текстового поля увеличится или уменьшится на заданную величину (рис. 10.38).

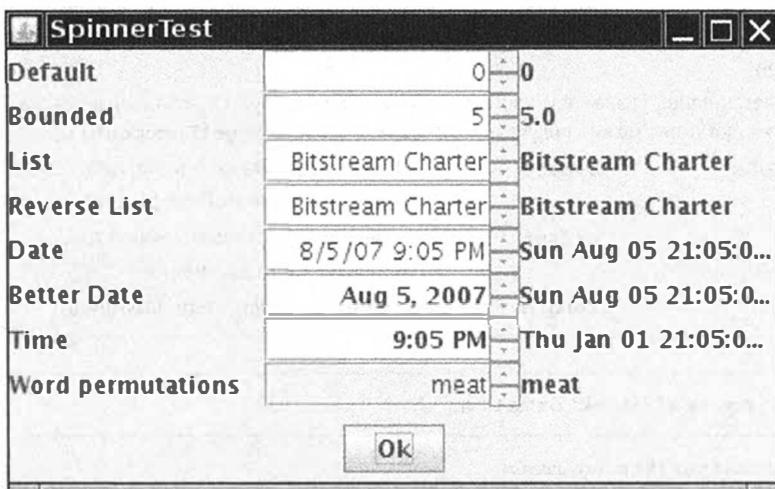


Рис. 10.38. Несколько вариантов исполнения компонента JSpinner

В качестве значений в счетчике могут использоваться числа, даты, значения, выбираемые из списка, или, как это чаще всего бывает, любая последовательность значений, для которых можно определить предшествующие и последующие значения. Для первых трех случаев в классе JSpinner определяются стандартные модели данных. Имеется также возможность определить собственную модель данных для описания произвольных последовательностей значений.

По умолчанию счетчик оперирует целыми числами, а его кнопки увеличивают или уменьшают целое число на 1. Текущее значение в счетчике можно получить, вызвав метод `getValue()`. Этот метод возвращает объект типа `Object`. Его нужно сначала привести к типу `Integer`, а затем извлечь упакованное значение, как показано ниже.

```
JSpinner defaultSpinner = new JSpinner();
. . .
int value = (Integer) defaultSpinner.getValue();
```

Исходную величину приращения в счетчике на 1 можно изменить, задав нижние и верхние пределы приращения. Ниже приведен пример реализации в коде счетчика с начальным значением 5, которое может изменяться в пределах от 0 и 10 с приращением 0,5.

```
JSpinner boundedSpinner =
    new JSpinner(new SpinnerNumberModel(5, 0, 10, 0.5));
```

Для класса `SpinnerNumberModel` предусмотрены два конструктора, причем один из них имеет только параметры типа `int`, а другой — только параметры типа `double`. Если любой из этих параметров принимает числовое значение

с плавающей запятой, то используется второй конструктор. Он выбирает объект типа `Double` для представления значений в счетчике.

Применение счетчиков не ограничивается только числовыми значениями. Имеется также возможность создать счетчик для перебора любого набора значений. Для этого достаточно передать модель типа `SpinnerListModel` конструктору класса `JSpinner`. Модель типа `SpinnerListModel` можно создать на основе любого массива или класса, реализующего интерфейс `List` (например, `ArrayList`). Так, в рассматриваемом здесь примере программы демонстрируется счетчик со всеми доступными именами шрифтов:

```
String[] fonts = GraphicsEnvironment.getLocalGraphicsEnvironment().getAvailableFontFamilyNames();
JSpinner listSpinner = new JSpinner(new SpinnerListModel(fonts));
```

Но направление приращения в счетчике оказалось нетипичным, поскольку оно отличалось от привычного для пользователей направления выбора значений в комбинированных списках. В комбинированном списке большие значения обычно располагаются ниже меньших значений, поэтому можно было вполне обоснованно ожидать, что после щелчка в счетчике на кнопке со стрелкой вниз будет выбрано большее значение. Но приращение счетчика происходит при увеличении индекса массива, а следовательно, большее значение получается после щелчка на кнопке со стрелкой вверх. Изменить направление приращения в модели типа `SpinnerListModel` нельзя, но желаемого результата можно все же добиться с помощью следующего анонимного подкласса:

```
JSpinner reverseListSpinner = new JSpinner(
    new SpinnerListModel(fonts)
{
    public Object getNextValue()
    {
        return super.getPreviousValue();
    }

    public Object getPreviousValue()
    {
        return super.getNextValue();
    }
});
```

Опробуйте оба варианта счетчика, чтобы выбрать наиболее интуитивный из них. Еще одно полезное применение счетчик находит для выбора дат. Создать такой счетчик с текущей датой в качестве начальной можно следующим образом:

```
JSpinner dateSpinner = new JSpinner(new SpinnerDateModel());
```

Но если посмотреть внимательно на рис. 10.38, то можно заметить, что текст в счетчике представляет как дату, так и время: **8/05/07 9:05 PM**. Для выбора даты время не так уж и важно. А для того чтобы счетчик показывал только дату, потребуется следующий код:

```
JSpinner betterDateSpinner = new JSpinner(new SpinnerDateModel());
String pattern = ((SimpleDateFormat)
    DateFormat.getDateInstance()).toPattern();
betterDateSpinner.setEditor(new JSpinner.DateEditor(
    betterDateSpinner, pattern));
```

Аналогичным образом можно выбирать из счетчика только время:

```
JSeparator timeSpinner = new JSeparator(new SpinnerDateModel());
pattern = ((SimpleDateFormat)
    DateFormat.getTimeInstance(DateFormat.SHORT)).toPattern();
timeSpinner.setEditor(new JSpinner.DateEditor(timeSpinner, pattern));
```

В счетчике можно отображать произвольные последовательности, определив собственную модель счетчика. В рассматриваемом здесь примере программы демонстрируется счетчик, предоставляющий все возможные варианты перестановки букв в слове "meat". Так, из него можно выбрать слова "mate", "meta", "team" и еще двадцать вариантов перестановки, щелкая на кнопках счетчика.

Для определения собственной модели счетчика придется расширить класс `AbstractSpinnerModel` и определить следующие четыре метода:

```
Object getValue()
void setValue(Object value)
Object getNextValue()
Object getPreviousValue()
```

Метод `getValue()` должен вызывать метод `fireStateChanged()` после установки нового значения. А метод `setValue()` устанавливает новое значение. Он должен генерировать исключение типа `IllegalArgumentException`, если новое значение оказывается неподходящим. Методы `getNextValue()` и `getPreviousValue()` возвращают значения, которые должны следовать после или перед текущим значением, или же пустое значение `null`, если достигнут конец последовательности.



Внимание! После установки нового значения метод `setValue()` должен вызвать метод `fireStateChanged()`. В противном случае поле счетчика не будет обновлено.



Внимание! Методы `getNextValue()` и `getPreviousValue()` не должны изменять текущее значение. Когда пользователь щелкает на кнопке счетчика со стрелкой вверх, вызывается метод `getNextValue()`. Если возвращаемое значение будет отличаться от пустого значения `null`, оно будет далее задано при вызове метода `setValue()`.

В рассматриваемом здесь примере программы для определения следующего и предыдущего вариантов перестановки букв в слове применяется стандартный алгоритм, как показано в листинге 10.24. Внутренний механизм работы этого алгоритма в данном случае не имеет особого значения. А в листинге 10.23 демонстрируется, каким образом создаются различные виды счетчиков. Чтобы увидеть значения в счетчиках, следует щелкнуть на кнопке ОК.

Листинг 10.23. Исходный код из файла spinner/SpinnerFrame.java

```
1 package spinner;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.text.*;
6 import javax.swing.*;
7
8 /**
```

```
9  * Фрейм с панелью, содержащей несколько счетчиков и
10 * кнопку для отображения значений в счетчиках
11 */
12 public class SpinnerFrame extends JFrame
13 {
14     private JPanel mainPanel;
15     private JButton okButton;
16
17     public SpinnerFrame()
18     {
19         JPanel buttonPanel = new JPanel();
20         okButton = new JButton("Ok");
21         buttonPanel.add(okButton);
22         add(buttonPanel, BorderLayout.SOUTH);
23
24         mainPanel = new JPanel();
25         mainPanel.setLayout(new GridLayout(0, 3));
26         add(mainPanel, BorderLayout.CENTER);
27         JSpinner defaultSpinner = new JSpinner();
28         addRow("Default", defaultSpinner);
29
30         JSpinner boundedSpinner =
31             new JSpinner(new SpinnerNumberModel(5, 0, 10, 0.5));
32         addRow("Bounded", boundedSpinner);
33
34         String[] fonts = GraphicsEnvironment.
35                     .getLocalGraphicsEnvironment()
36                     .getAvailableFontFamilyNames();
37
38         JSpinner listSpinner = new JSpinner(
39             new SpinnerListModel(fonts));
40         addRow("List", listSpinner);
41
42         JSpinner reverseListSpinner = new JSpinner(
43             new SpinnerListModel(fonts)
44             {
45                 public Object getNextValue()
46                     { return super.getPreviousValue(); }
47                 public Object getPreviousValue()
48                     { return super.getNextValue(); }
49             });
50         addRow("Reverse List", reverseListSpinner);
51
52         JSpinner dateSpinner = new JSpinner(new SpinnerDateModel());
53         addRow("Date", dateSpinner);
54
55         JSpinner betterDateSpinner =
56             new JSpinner(new SpinnerDateModel());
57         String pattern = ((SimpleDateFormat)
58             DateFormat.getDateInstance()).toPattern();
59         betterDateSpinner.setEditor(new JSpinner.DateEditor(
60             betterDateSpinner, pattern));
61         addRow("Better Date", betterDateSpinner);
62
63         JSpinner timeSpinner = new JSpinner(new SpinnerDateModel());
64         pattern = ((SimpleDateFormat) DateFormat
65             .getTimeInstance(DateFormat.SHORT)).toPattern();
66         timeSpinner.setEditor(new JSpinner.DateEditor(
67             timeSpinner, pattern));
68         addRow("Time", timeSpinner);
```

```

69
70     JSpinner permSpinner =
71         new JSpinner(new PermutationSpinnerModel("meat"));
72     addRow("Word permutations", permSpinner);
73     pack();
74 }
75
76 /**
77 * Ввести ряд счетчиков на основной панели
78 * @param labelText Метка счетчика
79 * @param spinner Образцовый счетчик
80 */
81 public void addRow(String labelText, final JSpinner spinner)
82 {
83     mainPanel.add(new JLabel(labelText));
84     mainPanel.add(spinner);
85     final JLabel valueLabel = new JLabel();
86     mainPanel.add(valueLabel);
87     okButton.addActionListener(new ActionListener()
88     {
89         public void actionPerformed(ActionEvent event)
90         {
91             Object value = spinner.getValue();
92             valueLabel.setText(value.toString());
93         }
94     });
95 }
96 }
```

Листинг 10.24. Исходный код из файла spinner/PermutationSpinnerModel.java

```

1 package spinner;
2
3 import javax.swing.*;
4
5 /**
6 * Модель, динамически формирующая перестановки слов
7 */
8 public class PermutationSpinnerModel extends AbstractSpinnerModel
9 {
10    private String word;
11
12    /**
13     * Конструирует модель
14     * @param w Переставляемое слово
15    */
16    public PermutationSpinnerModel(String w)
17    {
18        word = w;
19    }
20
21    public Object getValue()
22    {
23        return word;
24    }
25
26    public void setValue(Object value)
27    {
28        if (!(value instanceof String))
29        {
30            throw new IllegalArgumentException("Value must be a string");
31        }
32        word = value.toString();
33    }
34}
```

```
29         throw new IllegalArgumentException();
30     word = (String) value;
31     fireStateChanged();
32 }
33
34 public Object getNextValue()
35 {
36     int[] codePoints = toCodePointArray(word);
37
38     for (int i = codePoints.length - 1; i > 0; i--)
39     {
40         if (codePoints[i - 1] < codePoints[i])
41         {
42             int j = codePoints.length - 1;
43             while (codePoints[i - 1] > codePoints[j])
44                 j--;
45             swap(codePoints, i - 1, j);
46             reverse(codePoints, i, codePoints.length - 1);
47             return new String(codePoints, 0, codePoints.length);
48         }
49     }
50     reverse(codePoints, 0, codePoints.length - 1);
51     return new String(codePoints, 0, codePoints.length);
52 }
53
54 public Object getPreviousValue()
55 {
56     int[] codePoints = toCodePointArray(word);
57     for (int i = codePoints.length - 1; i > 0; i--)
58     {
59         if (codePoints[i - 1] > codePoints[i])
60         {
61             int j = codePoints.length - 1;
62             while (codePoints[i - 1] < codePoints[j])
63                 j--;
64             swap(codePoints, i - 1, j);
65             reverse(codePoints, i, codePoints.length - 1);
66             return new String(codePoints, 0, codePoints.length);
67         }
68     }
69     reverse(codePoints, 0, codePoints.length - 1);
70     return new String(codePoints, 0, codePoints.length);
71 }
72
73 private static int[] toCodePointArray(String str)
74 {
75     int[] codePoints =
76         new int[str.codePointCount(0, str.length())];
77     for (int i = 0, j = 0; i < str.length(); i++, j++)
78     {
79         int cp = str.codePointAt(i);
80         if (Character.isSupplementaryCodePoint(cp)) i++;
81         codePoints[j] = cp;
82     }
83     return codePoints;
84 }
85 private static void swap(int[] a, int i, int j)
86 {
87     int temp = a[i];
88     a[i] = a[j];
```

```

89     a[j] = temp;
90 }
91
92 private static void reverse(int[] a, int i, int j)
93 {
94     while (i < j)
95     {
96         swap(a, i, j);
97         i++;
98         j--;
99     }
100 }
101 }
```

`javax.swing.JSpinner 1.4`

- **`JSpinner()`**
Создает счетчик, редактирующий целочисленное значение, начиная с 0, с приращением 1 и без всяких ограничений.
- **`JSpinner(SpinnerModel model)`**
Создает счетчик, использующий указанную модель данных.
- **`Object getValue()`**
Получает текущее значение счетчика.
- **`void setValue(Object value)`**
Пытается установить значение счетчика. Генерирует исключение типа `IllegalArgumentException`, если модель не принимает это значение.
- **`void setEditor(JComponent editor)`**
Устанавливает компонент, используемый для редактирования значения счетчика.

`javax.swing.SpinnerNumberModel 1.4`

- **`SpinnerNumberModel(int initval, int minimum, int maximum, int stepSize)`**
- **`SpinnerNumberModel(double initval, double minimum, double maximum, double stepSize)`**
Эти конструкторы позволяют создавать модели, манипулирующие значениями типа `Integer` или `Double`. Для значений, не имеющих ограничений, следует использовать константы `MIN_VALUE` и `MAX_VALUE` из классов `Integer` и `Double`.

`javax.swing.SpinnerNumberModel 1.4 (окончание)`

Параметры:	<code>initval</code>	Исходное значение
	<code>minimum</code>	Минимально допустимое значение
	<code>maximum</code>	Максимально допустимое значение
	<code>stepSize</code>	Положительное или отрицательное приращение каждого счетчика

javax.swing.SpinnerListModel 1.4

- **SpinnerListModel (Object[] values)**
- **SpinnerListModel (List values)**

Эти конструкторы позволяют создавать модели для выбора среди заданных значений.

javax.swing.SpinnerDateModel 1.4

- **SpinnerDateModel ()**

Создает модель даты, в которой в качестве исходного значения служит текущая дата, отсутствуют верхние или нижние границы, а в качестве приращения принято значение константы `Calendar.DAY_OF_MONTH`.

- **SpinnerDateModel (Date initval, Comparable minimum, Comparable maximum, int step)**

Параметры:	initval	Исходное значение
	minimum	Минимально допустимое значение или пустое значение <code>null</code> , если нижняя граница не требуется
	maximum	Максимально допустимое значение или пустое значение <code>null</code> , если верхняя граница не требуется
	step	Дата, увеличивающаяся или уменьшающаяся при соответствующем приращении счетчика. Принимает значение одной из следующих констант из класса <code>Calendar</code> : <code>ERA</code> , <code>YEAR</code> , <code>MONTH</code> , <code>WEEK_OF_YEAR</code> , <code>WEEK_OF_MONTH</code> , <code>DAY_OF_MONTH</code> , <code>DAY_OF_YEAR</code> , <code>DAY_OF_WEEK</code> , <code>DAY_OF_WEEK_IN_MONTH</code> , <code>AM_PM</code> , <code>HOUR</code> , <code>HOUR_OF_DAY</code> , <code>MINUTE</code> , <code>SECOND</code> или <code>MILLISECOND</code>

java.text.SimpleDateFormat 1.1

- **String toPattern() 1.2**

Получает шаблон редактирования для данного средства форматирования даты. Обычно используется следующий шаблон: "`yyyy-MM-dd`". Подробнее об этом шаблоне см. в документации на Java SE.

javax.swing.JSpinner.DateEditor 1.4

- **DateEditor(JSpinner spinner, String pattern)**

Создает редактор даты для счетчика.

Параметры:

spinner

Счетчик, которому

принадлежит данный

редактор

pattern

Шаблон формата для

связанного с ним объекта

типа **SimpleDateFormat**

javax.swing.AbstractSpinnerModel 1.4

- **Object getValue()**

Получает текущее значение из модели счетчика.

- **void setValue(Object value)**

Пытается установить новое значение для модели счетчика. Генерирует исключение типа **IllegalArgumentException**, если значение неприемлемо. Переопределяя этот метод, следует вызывать метод **fireStateChanged()** после установки нового значения.

- **Object getNextValue()**

- **Object getPreviousValue()**

Вычисляют [но не устанавливают] последующее или предыдущее значение в последовательности, определяемой в модели счетчика.

10.4.4. Отображение HTML-документов средствами JEditorPane

В отличие от рассмотренных до сих пор текстовых компонентов, компонент **JEditorPane** позволяет отображать и редактировать текст в формате HTML и RTF. В частности, формат RTF (Rich Text Format — расширенный текстовый формат) применяется в приложениях корпорации Microsoft для обмена документами. Но этот формат недостаточно документирован и не вполне удовлетворительно работает даже в приложениях, выпускаемых корпорацией Microsoft, и поэтому здесь не рассматривается.

Функциональные возможности компонента **JEditorPane** ограничены. Средство воспроизведения данных в формате HTML способно отображать содержимое простых файлов, но не справляется с отображением сложных страниц, которые нередко встречаются в веб. К тому же этот компонент неустойчиво работает как HTML-редактор.

Наиболее приемлемое применение компоненту **JEditorPane** можно найти для отображения справочной информации в формате HTML. Содержимое справочных файлов приходится формировать самостоятельно, поэтому при отображении данных в формате HTML можно без особого труда обойти все ограничения, присущие компоненту **JEditorPane**.



На заметку! Чтобы получить более подробные сведения об инструментальных средствах для создания справочных систем промышленного образца, обращайтесь по адресу <http://javahelp.java.net>.

В листинге 10.25 представлен исходный код примера программы, в рабочем окне которой содержится панель редактирования, способная отображать HTML-страницы. Для этого в поле редактирования следует ввести URL, начинающийся с префикса `http:` или `file:`, а затем щелкнуть на кнопке Load (Загрузить). В итоге на панели редактирования появится указанная страница (рис. 10.39).

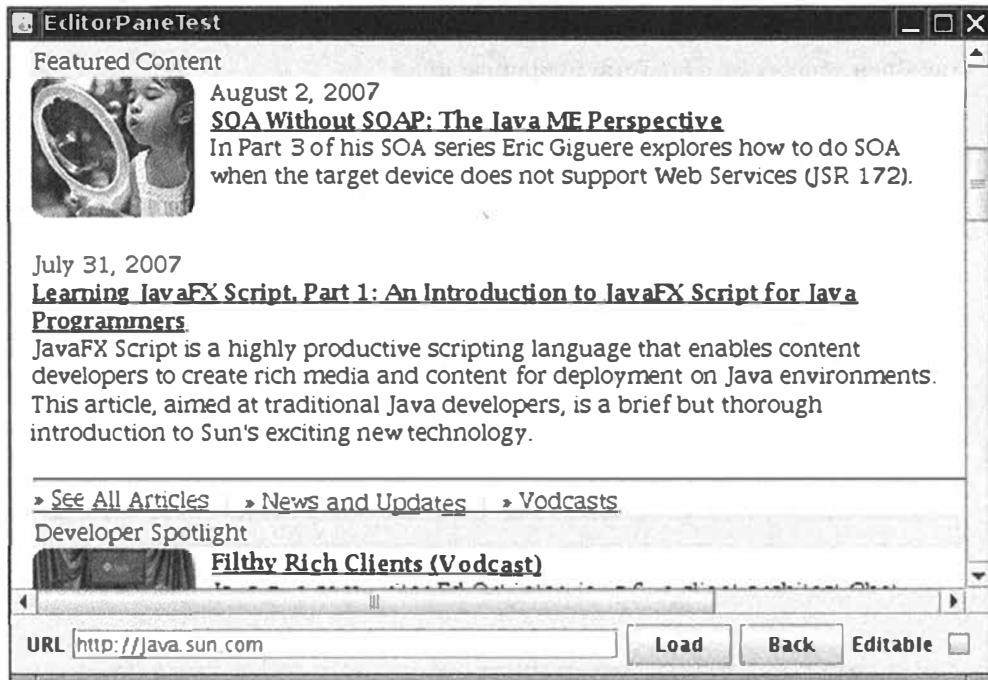


Рис. 10.39. Панель редактирования, на которой отображается HTML-страница

Все гипертекстовые ссылки на данной панели активны. Это означает, что если щелкнуть на любой ссылке, рассматриваемая здесь программа загрузит указанную веб-страницу. А если щелкнуть на кнопке Back (Назад), то на панели появится предыдущая веб-страница.

Данная программа, по существу, представляет собой простейший веб-браузер. Безусловно, в нем не поддерживаются такие специальные возможности, как кеширование страниц или список использованных ссылок, характерные для настоящих веб-браузеров. Кроме того, на панели редактирования этой программы нельзя отображать аплеты.

Если установить флажок Editable (Редактируемый), содержимое этой панели разрешается редактировать. Теперь на ней можно вводить текст, используя клавишу `<Backspace>` для удаления символов. Компонент `JEditorPane` способен также обрабатывать стандартные комбинации клавиш `<Ctrl+X>`, `<Ctrl+C>` и `<Ctrl+V>`,

которые выполняют операции вырезания, копирования и вставки текста. Но для того чтобы реализовать поддержку шрифтов и форматирование текста, придется затратить немало труда на программирование соответствующих функций.

При редактировании гипертекстовые ссылки в данном компоненте не активны. Кроме того, в режиме редактирования на некоторых веб-страницах будут видны команды сценария JavaScript, комментарии и другие дескрипторы (рис. 10.40). В рассматриваемом здесь примере программы режим редактирования разрешается использовать в учебных целях, но для реальных программ он не подходит.

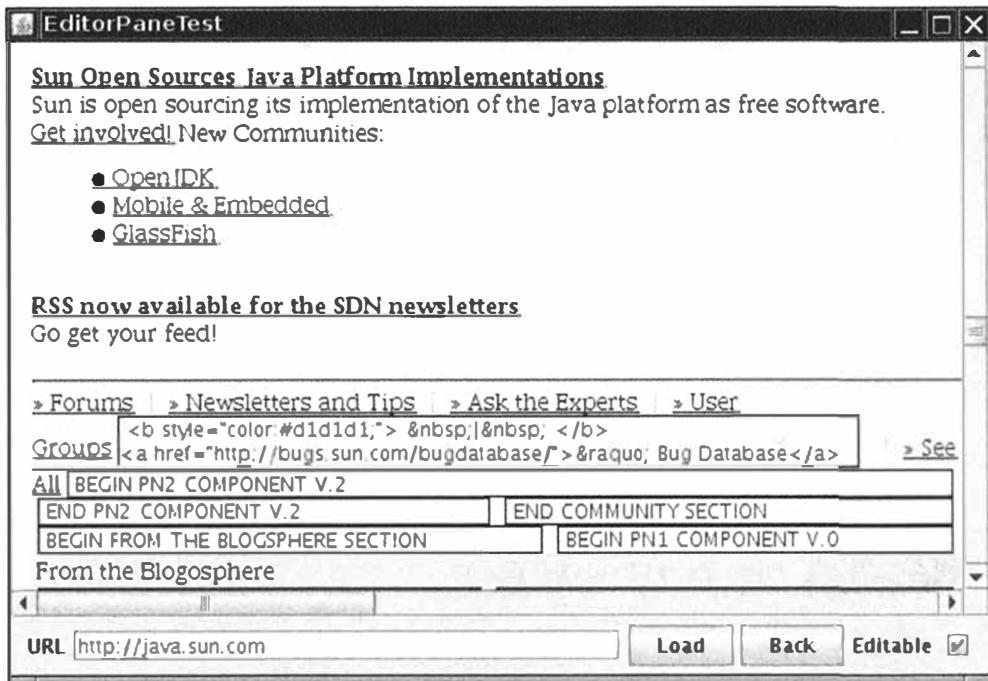


Рис. 10.40. Панель, действующая в режиме редактирования



Совет! По умолчанию компонент `JEditorPane` находится в режиме редактирования, а для отключения этого режима следует вызвать метод `editorPane.setEditable(false)`.

Основные функции панели редактирования в рассматриваемом здесь примере программы достаточно просты. Так, для загрузки документов вызывается метод `setPage()` с параметром в виде символьной строки, содержащей URL, или объекта типа URL:

```
JEditorPane editorPane = new JEditorPane();
editorPane setPage(url);
```

Класс `JEditorPane` расширяет класс `JTextComponent`. Поэтому для отображения обычного текста можно вызвать метод `setText()` из его суперкласса.



Совет! Из документации на прикладной программный интерфейс API трудно понять, загружает ли метод `setPage()` новый документ в отдельном потоке исполнения, что было бы очень кстати, поскольку компонент `JEditorPane` нельзя назвать сверхбыстро действующим. Чтобы явно указать на необходимость загрузки документа в отдельном потоке исполнения, можно воспользоваться следующим фрагментом кода:

```
AbstractDocument doc = (AbstractDocument) editorPane.getDocument();
doc.setAsynchronousLoadPriority(0);
```

Для обработки событий от щелчков кнопкой мыши на гипертекстовых ссылках следует ввести соответствующий обработчик типа `HyperlinkListener`. У интерфейса `HyperlinkListener` имеется единственный метод `hyperlinkUpdate()` с параметром типа `HyperlinkEvent`. Этот метод вызывается при наведении курсора и щелчке на гипертекстовой ссылке. А для определения вида наступившего события следует вызвать метод `getEventType()`. Он возвращает значение одной из перечисленных ниже констант.

```
HyperlinkEvent.EventType.ACTIVATED
HyperlinkEvent.EventType.ENTERED
HyperlinkEvent.EventType.EXITED
```

Константа `HyperlinkEvent.EventType.ACTIVATED` обозначает щелчок кнопкой мыши на гипертекстовой ссылке. В этом случае обычно требуется открыть указанную веб-страницу. А константы `HyperlinkEvent.EventType.ENTERED` и `HyperlinkEvent.EventType.EXITED` обычно служат для организации визуальной обратной связи, например, для изменения внешнего вида ссылки в тот момент, когда пользователь наводит на нее курсор мыши.



На заметку! Не меньшей загадкой остается и то, почему разработчики библиотеки Swing не предусмотрели в интерфейсе `HyperlinkListener` три отдельных метода для обработки событий, связанных с активизацией гипертекстовой ссылки, наведением и отведением курсора от ссылки.

Метод `getURL()` из класса `HyperlinkEvent` возвращает URL по выбранной гипертекстовой ссылке. Ниже приведен пример установки обработчика событий, наступающих при активизации пользователем гипертекстовых ссылок. Этот обработчик событий извлекает URL и обновляет панель редактирования. Метод `setPage()` может генерировать исключение типа `IOException`. В этом случае сообщение об ошибке выводится обычным текстом.

```
editorPane.addHyperlinkListener(new
    HyperlinkListener()
{
    public void hyperlinkUpdate(HyperlinkEvent event)
    {
        if (event.getEventType() == HyperlinkEvent.EventType.ACTIVATED)
        {
            try
            {
                editorPane.setPage(event.getURL());
            }
            catch (IOException e)
            {
                editorPane.setText("Exception: " + e);
            }
        }
    }
});
```

```
    }  
    }  
});
```

В примере программы из листинга 10.25 демонстрируются все основные функции панели редактирования, необходимые для создания справочной системы с выводом текста в формате HTML. Компонент JEditorPane имеет более сложную внутреннюю структуру, чем компоненты построения таблиц или деревьев. Но сложности устройства его внутреннего механизма скрыты, что не позволяет создавать на его основе текстовые редакторы и средства воспроизведения текста в специальных форматах.

Листинг 10.25. Исходный текст из файла editorPane/EditorPaneFrame.java

```
1 package editorPane;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.io.*;
6 import java.util.*;
7 import javax.swing.*;
8 import javax.swing.event.*;
9
10 /**
11 * Этот фрейм содержит панель редактирования, текстовое поле и
12 * кнопку для ввода URL и загрузки документа, а также кнопку
13 * для возврата к предыдущему загруженному документу
14 */
15 public class EditorPaneFrame extends JFrame
16 {
17     private static final int DEFAULT_WIDTH = 600;
18     private static final int DEFAULT_HEIGHT = 400;
19
20     public EditorPaneFrame()
21     {
22         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
23
24         final Stack<String> urlStack = new Stack<>();
25         final JEditorPane editorPane = new JEditorPane();
26         final JTextField url = new JTextField(30);
27
28         // установить приемник событий от гиперссылок
29
30         editorPane.setEditable(false);
31         editorPane.addHyperlinkListener(event ->
32         {
33             if (event.getEventType() ==
34                 HyperlinkEvent.EventType.ACTIVATED)
35             {
36                 try
37                 {
38                     // запомнить URL для кнопки возврата
39                     urlStack.push(event.getURL().toString());
40                     // показать URL в текстовом поле
41                     url.setText(event.getURL().toString());
42                     editorPane.setPage(event.getURL());
43                 }
44             }
45         });
46
47         getContentPane().add(editorPane, "Center");
48         getContentPane().add(url, "South");
49
50         pack();
51     }
52 }
```

```
43         }
44         catch (IOException e)
45         {
46             editorPane.setText("Exception: " + e);
47         }
48     });
49 });
50
51 // настроить флагок для переключения режима редактирования
52
53 final JCheckBox editable = new JCheckBox();
54 editable.addActionListener(event ->
55 editorPane.setEditable(editable.isSelected()));
56
57 // настроить кнопку для загрузки документа по заданному URL
58
59 ActionListener listener = event ->
60 {
61     try
62     {
63         // запомнить URL для кнопки возврата
64         urlStack.push(url.getText());
65         editorPane setPage(url.getText());
66     }
67     catch (IOException e)
68     {
69         editorPane.setText("Exception: " + e);
70     }
71 };
72
73 JButton loadButton = new JButton("Load");
74 loadButton.addActionListener(listener);
75 url.addActionListener(listener);
76
77 // настроить кнопку возврата и ее действие
78
79 JButton backButton = new JButton("Back");
80 backButton.addActionListener(event ->
81 {
82     if (urlStack.size() <= 1) return;
83     try
84     {
85         // получить URL из кнопки возврата
86         urlStack.pop();
87         // показать URL в текстовом поле
88         String urlString = urlStack.peek();
89         url.setText(urlString);
90         editorPane setPage(urlString);
91     }
92     catch (IOException e)
93     {
94         editorPane.setText("Exception: " + e);
95     }
96 });
97
98 add(new JScrollPane(editorPane), BorderLayout.CENTER);
99
100 // разместить все элементы управления на панели
101 JPanel panel = new JPanel();
```

```

103     panel.add(new JLabel("URL"));
104     panel.add(url);
105     panel.add(loadButton);
106     panel.add(backButton);
107     panel.add(new JLabel("Editable"));
108     panel.add(editable);
109
110     add(panel, BorderLayout.SOUTH);
111 }
112 }
```

javax.swing.JEditorPane 1.2

- **void setPage(URL url)**

Загружает на панели редактирования веб-страницу по указанному URL.

- **void addHyperlinkListener(HyperLinkListener listener)**

Добавляет приемник событий от гипертекстовых ссылок на панели редактирования.

javax.swing.event.HyperlinkListener 1.2

- **void hyperlinkUpdate(HyperlinkEvent event)**

Вызывается всякий раз, когда выбирается гипертекстовая ссылка.

javax.swing.event.HyperlinkEvent 1.2

- **URL getURL()**

Возвращает URL по выбранной гипертекстовой ссылке.

10.5. Индикаторы состояния

В этом разделе рассматриваются три класса, предназначенные для отображения хода выполнения длительных заданий и процессов. В частности, класс `JProgressBar` — это компонент `Swing`, наглядно показывающий ход выполнения процесса. Класс `ProgressMonitor` поддерживает диалоговое окно, в котором находится индикатор выполнения, отображаемый с помощью объекта типа `JProgressBar`. А класс `ProgressMonitorInputStream` отображает диалоговое окно контроля текущего состояния при чтении данных из потока ввода.

10.5.1. Индикаторы выполнения

Индикатор выполнения — это широкая прямоугольная полоса, частично заполняемая другим цветом. Такой индикатор дает наглядное представление о ходе выполнения операции. По умолчанию в пределах прямоугольной полосы отображается также символьная строка в формате `п%`, где `п` — число. Пример рабочего окна, в нижней части которого находится индикатор выполнения, приведен на рис. 10.41.

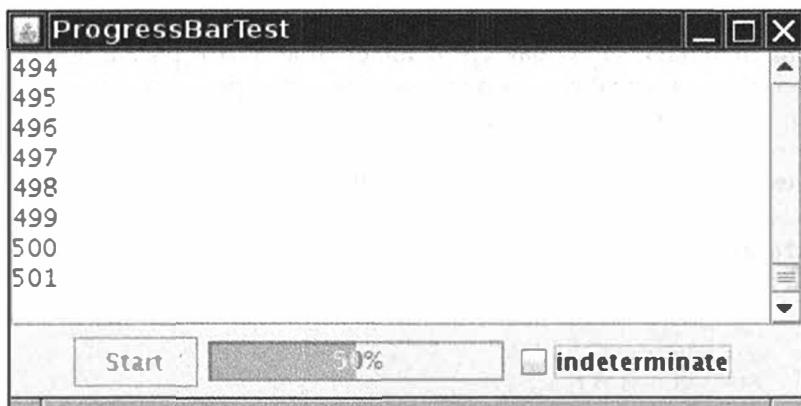


Рис. 10.41. Окно с индикатором выполнения

Индикатор выполнения создается подобно ползунковому регулятору. Для этого конструктору класса `JProgressBar` передается минимальное и максимальное значения, а дополнительно, хотя и необязательно, — ориентация данного компонента:

```
progressBar = new JProgressBar(0, 1000);
progressBar = new JProgressBar(SwingConstants.VERTICAL, 0, 1000);
```

Минимальное и максимальное значения, отображаемые в индикаторе выполнения, можно также задать с помощью методов `setMinimum()` и `setMaximum()`. В отличие от ползункового регулятора, индикатор выполнения не реагирует на действия пользователя. Чтобы изменить его состояние, следует вызвать метод `setValue()`.

Если вызвать метод `progressBar.setStringPainted(true)`, то в индикаторе будет рассчитана доля выполнения операции в процентах и отображена символьная строка в формате `##%`. Если же требуется вывести другую последовательность символов, то соответствующую строку следует передать методу `setString()`:

```
if (progressBar.getValue() > 900)
    progressBar.setString("Almost Done");
```

В примере программы, исходный код которой приведен в листинге 10.26, демонстрируется индикатор выполнения, предназначенный для визуального контроля над ходом имитируемого длительного процесса. В классе `SimulatedActivity` значение переменной `current` увеличивается 10 раз в секунду. Когда оно достигает конечной величины, задаваемой в переменной `target`, выполнение имитируемого процесса завершается. В данной программе класс `SwingWorker` используется с целью симитировать процесс и обновить индикатор его выполнения в методе `process()`. Этот метод вызывается в классе `SwingWorker` из потока диспетчеризации событий, что дает возможность безопасно обновлять индикатор выполнения. (Более подробно вопросы обеспечения безопасности потоков исполнения в Swing рассматриваются в главе 14 первого тома настоящего издания.)

В версии JDK 1.4 был внедрен индикатор, уведомляющий о самом факте выполнения процесса. Он обеспечивает вывод анимационного изображения, но не отображает никаких сведений, позволяющих судить о том, какая именно часть

процесса выполнена. Подобного рода индикаторы часто применяются в браузерах. Они сообщают, что браузер ожидает ответа от сервера, но не дают ясного представления, сколько это ожидание может продлиться. Для отображения такого индикатора “неопределенного ожидания” следует вызвать метод `setIndeterminate()`. Весь исходный код примера программы, демонстрирующей применение индикатора выполнения, приведен в листинге 10.26.

Листинг 10.26. Исходный код из файла progressBar/ProgressBarFrame.java

```
1 package progressBar;
2
3 import java.awt.*;
4 import java.util.List;
5
6 import javax.swing.*;
7
8 /**
9  * Фрейм, содержащий кнопку для запуска имитируемого процесса,
10 * индикатор выполнения и текстовую область для вывода результатов
11 */
12 public class ProgressBarFrame extends JFrame
13 {
14     public static final int TEXT_ROWS = 10;
15     public static final int TEXT_COLUMNS = 40;
16
17     private JButton startButton;
18     private JProgressBar progressBar;
19     private JCheckBox checkBox;
20     private JTextArea textArea;
21     private SimulatedActivity activity;
22
23     public ProgressBarFrame()
24     {
25         // в этой текстовой области выводятся результаты
26         // выполнения имитируемого процесса
27         textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
28
29         // установить панель с кнопкой и индикатором выполнения
30
31         final int MAX = 1000;
32         JPanel panel = new JPanel();
33         startButton = new JButton("Start");
34         progressBar = new JProgressBar(0, MAX);
35         progressBar.setStringPainted(true);
36         panel.add(startButton);
37         panel.add(progressBar);
38
39         checkBox = new JCheckBox("indeterminate");
40         checkBox.addActionListener(event ->
41             {
42                 progressBar.setIndeterminate(checkBox.isSelected());
43                 progressBar.setStringPainted(
44                     !progressBar.isIndeterminate());
45             });
46         panel.add(checkBox);
47         add(new JScrollPane(textArea), BorderLayout.CENTER);
```

```
48     add(panel, BorderLayout.SOUTH);
49
50     // установить приемник действий кнопки
51
52     startButton.addActionListener(event ->
53     {
54         startButton.setEnabled(false);
55         activity = new SimulatedActivity(MAX);
56         activity.execute();
57     });
58     pack();
59 }
60
61 class SimulatedActivity extends SwingWorker<Void, Integer>
62 {
63     private int current;
64     private int target;
65
66     /**
67      * Имитирует процесс инкрементирования счетчика от нуля
68      * до заданного конечного значения
69      * @param t the target value of the counter
70      */
71     public SimulatedActivity(int t)
72     {
73         current = 0;
74         target = t;
75     }
76
77     protected Void doInBackground() throws Exception
78     {
79         try
80         {
81             while (current < target)
82             {
83                 Thread.sleep(100);
84                 current++;
85                 publish(current);
86             }
87         }
88         catch (InterruptedException e)
89         {
90         }
91         return null;
92     }
93
94     protected void process(List<Integer> chunks)
95     {
96         for (Integer chunk : chunks)
97         {
98             textArea.append(chunk + "\n");
99             progressBar.setValue(chunk);
100        }
101
102    protected void done()
103    {
104        startButton.setEnabled(true);
105    }
106 }
```

```
105     }
106   }
107 }
```

10.5.2. Мониторы текущего состояния

Индикатор выполнения является простым компонентом, размещаемым в окне. А класс `ProgressMonitor` реализует полностью диалоговое окно монитора текущего состояния, содержащее индикатор выполнения (рис. 10.42), а также кнопку `Cancel`. Если пользователь щелкнет на этой кнопке, диалоговое окно монитора закроется. Кроме того, программа, выполнение которой отслеживается монитором, может проверить, закрыто ли диалоговое окно, и завершить свою работу, если пользователь выбрал кнопку `Cancel`. (Обратите внимание на то, что имя класса, реализующего данный компонент, не начинается с буквы `J`.)

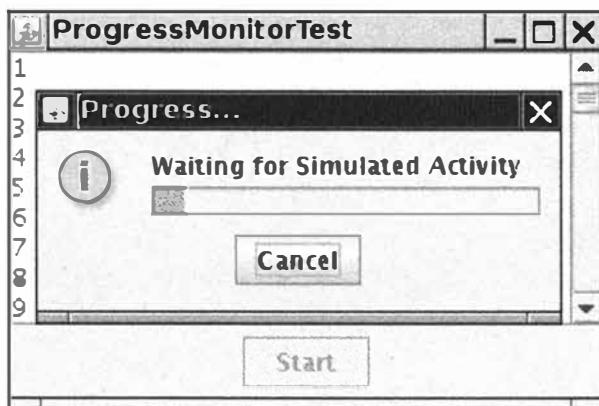


Рис. 10.42. Диалоговое окно монитора текущего состояния

Для создания монитора текущего состояния конструктору класса `ProgressMonitor` передаются следующие данные.

- Родительский компонент, поверх которого должно отображаться диалоговое окно монитора.
- Объект (символьная строка, пиктограмма или компонент), отображаемый в диалоговом окне монитора.
- Необязательное примечание, которое выводится ниже данного объекта.
- Минимальное и максимальное значения.

Но монитор текущего состояния не может самостоятельно определить, какая именно часть процесса выполнена, или завершить его. Поэтому состояние диалогового окна монитора необходимо периодически обновлять, вызывая метод `setProgress()`. (Этот метод выполняет те же самые функции, что и метод `setValue()` из класса `JProgressBar`.) Когда контролируемый монитором процесс благополучно завершится, следует вызвать метод `close()`, удалив диалоговое окно монитора с экрана. Впоследствии то же самое окно можно использовать повторно, вызвав метод `start()`.

Самые большие трудности, возникающие при использовании диалогового окна монитора текущего состояния, связаны с обработкой запросов на отмену контролируемого процесса. Дело в том, что к кнопке Cancel нельзя присоединить соответствующий обработчик событий. Вместо этого приходится периодически вызывать метод `isCanceled()`, чтобы выяснить, щелкнул ли пользователь программы на кнопке Cancel.

Если рабочий поток исполнения может входить в состояние блокировки на неопределенный срок (например, при чтении введенных данных через сетевое соединение), он не в состоянии следить за кнопкой Cancel. В рассматриваемом здесь примере программы демонстрируется, каким образом для этой цели используется таймер. Кроме того, на таймер возлагается ответственность за обновление замеров текущего состояния и хода выполнения процесса.

Если запустить рассматриваемую здесь программу, исходный код которой приведен в листинге 10.27, то в ходе ее выполнения обнаружится интересная особенность. Диалоговое окно монитора не исчезает, как только завершится контролируемый процесс. Программе нужно сначала убедиться, что имитируемый процесс действительно завершился или, скорее всего, завершился раньше того времени, которое требуется для отображения диалогового окна.

Управление выдержкой времени происходит следующим образом. С помощью метода `setMillisToDecideToPopUp()` задается число миллисекунд, которые должны пройти между созданием объекта диалогового окна и принятием решения относительно его отображения. По умолчанию принимается значение 500 миллисекунд. Метод `setMillisToPopUp()` служит для оценки времени, требующегося для отображения диалогового окна. Разработчики библиотек Swing установили это значение равным по умолчанию 2 секундам. Они учли тот факт, что диалоговые окна компонентов Swing не появляются так быстро, как хотелось бы. Поэтому изменять это значение без особой надобности не рекомендуется.

Листинг 10.27. Исходный код из файла `progressMonitor/ProgressMonitorFrame.java`

```
1 package progressMonitor;
2
3 import java.awt.*;
4
5 import javax.swing.*;
6
7 /**
8 * Фрейм, содержащий кнопку для запуска имитируемого процесса и
9 * текстовое поле для вывода результатов выполнения этого процесса
10 */
11 class ProgressMonitorFrame extends JFrame
12 {
13     public static final int TEXT_ROWS = 10;
14     public static final int TEXT_COLUMNS = 40;
15
16     private Timer cancelMonitor;
17     private JButton startButton;
18     private ProgressMonitor progressDialog;
19     private JTextArea textArea;
20     private SimulatedActivity activity;
```

```
21 public ProgressMonitorFrame()
22 {
23     // в этой текстовой области выводятся результаты
24     // выполнения имитируемого процесса
25     textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
26
27     // установить панель с кнопками
28     JPanel panel = new JPanel();
29     startButton = new JButton("Start");
30     panel.add(startButton);
31
32     add(new JScrollPane(textArea), BorderLayout.CENTER);
33     add(panel, BorderLayout.SOUTH);
34
35     // установить приемник действий кнопки
36     startButton.addActionListener(event ->
37     {
38         startButton.setEnabled(false);
39         final int MAX = 1000;
40
41         // начать имитируемый процесс
42         activity = new SimulatedActivity(MAX);
43         activity.execute();
44
45         // открыть диалоговое окно монитора текущего состояния
46         progressDialog = new ProgressMonitor(
47             ProgressMonitorFrame.this,
48             "Waiting for Simulated Activity", null, 0, MAX);
49         cancelMonitor.start();
50     });
51
52     // установить приемник действий таймера
53
54     cancelMonitor = new Timer(500, event ->
55     {
56         if (progressDialog.isCanceled())
57         {
58             activity.cancel(true);
59             startButton.setEnabled(true);
60         }
61         else if (activity.isDone())
62         {
63             progressDialog.close();
64             startButton.setEnabled(true);
65         }
66         else
67         {
68             progressDialog.setProgress(activity.getProgress());
69         }
70     });
71     pack();
72 }
73
74 class SimulatedActivity extends SwingWorker<Void, Integer>
75 {
76     private int current;
77     private int target;
```

```
81     /**
82      * Имитирует процесс инкрементирования счетчика от нуля
83      * до заданного конечного значения
84      * @param t Конечное значение счетчика
85      */
86     public SimulatedActivity(int t)
87     {
88         current = 0;
89         target = t;
90     }
91
92     protected Void doInBackground() throws Exception
93     {
94         try
95         {
96             while (current < target)
97             {
98                 Thread.sleep(100);
99                 current++;
100                textArea.append(current + "\n");
101                setProgress(current);
102            }
103        } catch (InterruptedException e)
104        {
105        }
106    }
107    return null;
108 }
109 }
110 }
```

10.5.3. Контроль процесса чтения данных из потока ввода

В состав библиотеки Swing входит фильтр типа `ProgressMonitorInputStream`, который автоматически отображает диалоговое окно, в котором контролируется процесс чтения данных из потока ввода. Пользоваться фильтром типа `ProgressMonitorInputStream` очень просто. Достаточно ввести его в обычную последовательность фильтруемых потоков ввода-вывода (подробнее о потоках ввода-вывода см. в главе 2). Допустим, требуется прочитать текст из файла. Для этого сначала открывается поток ввода типа `FileInputStream`, как показано ниже.

```
FileInputStream in = new FileInputStream(f);
```

Обычно этот поток преобразуется в поток ввода типа `InputStreamReader` следующим образом:

```
InputStreamReader reader = new InputStreamReader(in);
```

Но, для того чтобы контролировать чтение данных из потока ввода, нужно сначала преобразовать этот поток в поток ввода с контролем текущего состояния:

```
ProgressMonitorInputStream progressIn =
    new ProgressMonitorInputStream(parent, caption, in);
```

Конструктору класса `ProgressMonitorInputStream` передаются родительский компонент, заголовок и сам контролируемый поток ввода. В методе `read()` из этого класса анализируется количество прочитанных байтов и обновляется содержимое диалогового окна с монитором текущего состояния потока ввода.

Далее можно продолжить формирование последовательности фильтруемых потоков, как показано в приведенной ниже строке кода.

```
InputStreamReader reader = new InputStreamReader(progressIn);
```

Вот, собственно, и все, что требуется для организации контроля над чтением данных из потока ввода. При чтении из файла автоматически открывается диалоговое окно, показанное на рис. 10.43. Это очень изящное применение фильтрации потоков ввода-вывода.

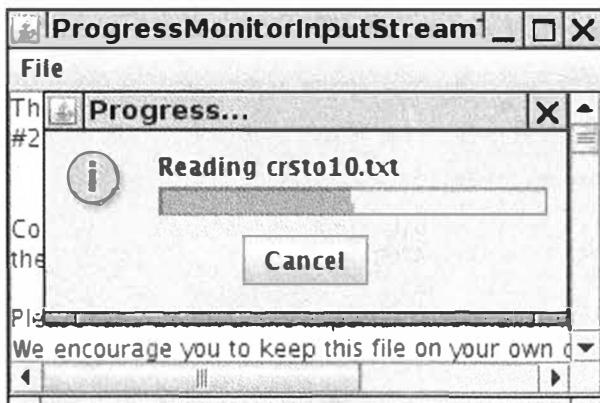


Рис.10.43. Окно, в котором контролируется чтение данных из потока ввода

На заметку! Для определения общего количества байтов в потоке ввода с контролем текущего состояния служит метод `available()` из класса `InputStream`. Но этот метод возвращает лишь количество байтов, которые могут быть прочитаны из потока ввода без блокировки. Таким образом, рассматриваемый здесь монитор текущего состояния потоков ввода вполне пригоден для контроля над чтением данных из файлов и источников в Интернете по сетевому протоколу HTTP, поскольку объем этих данных заранее известен. Но, к сожалению, это правило распространяется далеко не на все потоки ввода-вывода.

В примере программы, исходный код которой приведен в листинге 10.28, подсчитывается количество строк в файле. Если файл крупный, то открывается диалоговое окно с монитором текущего состояния процесса чтения данных из файла. Если же пользователь щелкнет на кнопке `Cancel`, поток ввода закрывается. А поскольку коду, обрабатывающему вводимые данные, заранее известно, что нужно делать по окончании их ввода, то для правильной обработки состояния отмены данного процесса дополнительная логика не требуется.

Однако в данной программе применяется далеко не самый эффективный способ заполнения текстовой области. Эта программа работала бы намного быстрее, если бы в ней можно было прочитать сначала файл в объект типа `StringBuilder`, а затем связать текстовую область с содержимым этого объекта. Но в данном случае медленное выполнение программы вполне уместно, поскольку оно предоставляет достаточно времени, чтобы не спеша оценить выполнение процесса в диалоговом окне. Во избежание мерцания текстовая область не отображается до ее заполнения.

Листинг 10.28. Исходный код из файла progressMonitorInputStream/
TextFrame.java

```
1 package progressMonitorInputStream;
2
3 import java.io.*;
4 import java.nio.file.*;
5 import java.util.*;
6
7 import javax.swing.*;
8
9 /**
10 * Фрейм с меню для загрузки текстового файла и текстовой
11 * областью для отображения его содержимого. Эта область
12 * создается при загрузке файла и устанавливается в виде
13 * панели содержимого фрейма по завершении загрузки во избежание
14 * мерцания в ходе данного процесса
15 */
16 public class TextFrame extends JFrame
17 {
18     public static final int TEXT_ROWS = 10;
19     public static final int TEXT_COLUMNS = 40;
20
21     private JMenuItem openItem;
22     private JMenuItem exitItem;
23     private JTextArea textArea;
24     private JFileChooser chooser;
25
26     public TextFrame()
27     {
28         textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
29         add(new JScrollPane(textArea));
30
31         chooser = new JFileChooser();
32         chooser.setCurrentDirectory(new File("."));
33
34         JMenuBar menuBar = new JMenuBar();
35         setJMenuBar(menuBar);
36         JMenu fileMenu = new JMenu("File");
37         menuBar.add(fileMenu);
38         openItem = new JMenuItem("Open");
39         openItem.addActionListener(event ->
40             {
41                 try
42                 {
43                     openFile();
44                 }
45                 catch (IOException exception)
46                 {
47                     exception.printStackTrace();
48                 }
49             });
50
51         fileMenu.add(openItem);
52         exitItem = new JMenuItem("Exit");
53         exitItem.addActionListener(event -> System.exit(0));
54         fileMenu.add(exitItem);
55         pack();
56     }
}
```

```

57 /**
58  * Приглашает пользователя выбрать файл, загружает его
59  * в текстовую область и устанавливает ее в виде панели
60  * содержимого фрейма
61 */
62 public void openFile() throws IOException
63 {
64     int r = chooser.showOpenDialog(this);
65     if (r != JFileChooser.APPROVE_OPTION) return;
66     final File f = chooser.getSelectedFile();
67
68     // установить последовательность потоков ввода и
69     // фильтрации читаемых данных
70
71     InputStream fileIn = Files.newInputStream(f.toPath());
72     final ProgressMonitorInputStream progressIn =
73         new ProgressMonitorInputStream(this, "Reading "
74             + f.getName(), fileIn);
75
76     textArea.setText("");
77
78     SwingWorker<Void, Void> worker =
79         new SwingWorker<Void, Void>()
80     {
81         protected Void doInBackground() throws Exception
82         {
83             try (Scanner in = new Scanner(progressIn, "UTF-8"))
84             {
85                 while (in.hasNextLine())
86                 {
87                     String line = in.nextLine();
88                     textArea.append(line);
89                     textArea.append("\n");
90                 }
91             }
92             return null;
93         }
94     };
95     worker.execute();
96 }
97 }
98 }
```

javax.swing.JProgressBar 1.2

- **JProgressBar()**
- **JProgressBar(int direction)**
- **JProgressBar(int min, int max)**
- **JProgressBar(int direction, int min, int max)**

Конструируют индикатор выполнения с указанной ориентацией, а также минимальным и максимальным значениями.

Параметры:

direction

Принимает значение одной из
следующих констант:

javax.swing.JProgressBar 1.2 (окончание)***min, max***

SwingConstants.HORIZONTAL или **SwingConstants.VERTICAL**. По умолчанию выбирается горизонтальная ориентация. Минимальное и максимальное значения для индикатора выполнения. По умолчанию принимаются значения 0 и 100 соответственно.

- **int getMinimum()**
- **int getMaximum()**
- **void setMinimum(int value)**
- **void setMaximum(int value)**

Получают или устанавливают минимальное или максимальное значение.

- **int getValue()**
- **void setValue(int value)**

Получают или устанавливают текущее значение.

- **String getString()**
- **void setString(String s)**

Получают или устанавливают символьную строку, отображаемую на полосе индикатора выполнения. Если параметр **s** принимает пустое значение **null**, то отображается символьная строка, задаваемая по умолчанию в формате №.

- **boolean isStringPainted()**
- **void setStringPainted(boolean b)**

Получают или устанавливают значение свойства, определяющего порядок отображения символьной строки. Если это свойство принимает логическое значение **true**, символьная строка отображается на полосе индикатора выполнения. По умолчанию устанавливается логическое значение **false**, запрещающее отображение символьной строки.

- **boolean isIndeterminate() 1.4**
- **void setIndeterminate(boolean b) 1.4**

Получают или устанавливают значение свойства определяющего неопределенное состояние индикатора выполнения. Если это свойство принимает логическое значение **true**, индикатор выполнения превращается в прямоугольный блок, перемещающийся вперед и назад, указывая на неопределенное время ожидания. По умолчанию устанавливается логическое значение **false**.

javax.swing.ProgressMonitor 1.2

- **ProgressMonitor(Component parent, Object message, String note, int min, int max)**

Создает диалоговое окно монитора текущего состояния.

`javax.swing.ProgressMonitor` 1.2 (окончание)

Параметры:	parent	Родительский компонент, над которым отображается данное диалоговое окно
	message	Объект сообщения, отображаемый в диалоговом окне
	note	Необязательная строка примечания, выводимая ниже сообщения. Если данный параметр принимает пустое значение <code>null</code> , место для строки примечания не выделяется и последующие вызовы метода <code>setNote()</code> ничего не дают
	min, max	Минимальное и максимальное значения для индикатора выполнения

- **void setNote(String note)**
Изменяет текст примечания.
- **void setProgress(int value)**
Устанавливает указанное значение для индикатора выполнения.
- **void close()**
Закрывает диалоговое окно.
- **boolean isCanceled()**
Возвращает логическое значение `true`, если пользователь закрыл диалоговое окно, щелкнув на кнопке `Cancel`.

`javax.swing.ProgressMonitorInputStream` 1.2

• ProgressMonitorInputStream(Component parent, Object message, InputStream in)	Создает фильтр для потока ввода и связывает с ним диалоговое окно монитора текущего состояния.	
Параметры:	parent	Родительский компонент, над которым отображается данное диалоговое окно
	message	Объект сообщения, отображаемый в диалоговом окне
	in	Контролируемый поток ввода

10.6. Организаторы и декораторы компонентов

Обсуждение расширенных средств библиотеки `Swing` завершается рассмотрением компонентов, помогающих организовать другие компоненты ГПИ. К их числу относятся *разделяемые панели*, разбивающие область окна на несколько частей с регулируемыми границами, *панели с вкладками*, на которых пользователь может переходить от одной вкладки к другой, а также *настольные панели*, на которых можно размещать несколько *внутренних фреймов*. И в заключение будут

рассмотрены слои — декораторы, накладываемые на другие компоненты ГПИ для их специального оформления.

10.6.1. Разделяемые панели

Разделяемая панель позволяет разбивать компонент на части с регулируемыми границами. На рис. 10.44 приведен фрейм с двумя вложенными разделяемыми панелями. Компоненты на внешней разделяемой панели располагаются по вертикали: текстовая область — снизу, а другая, внутренняя разделяемая панель, — сверху. В свою очередь, компоненты на внутренней разделяемой панели располагаются по горизонтали: список — слева, а метка с изображением — справа.

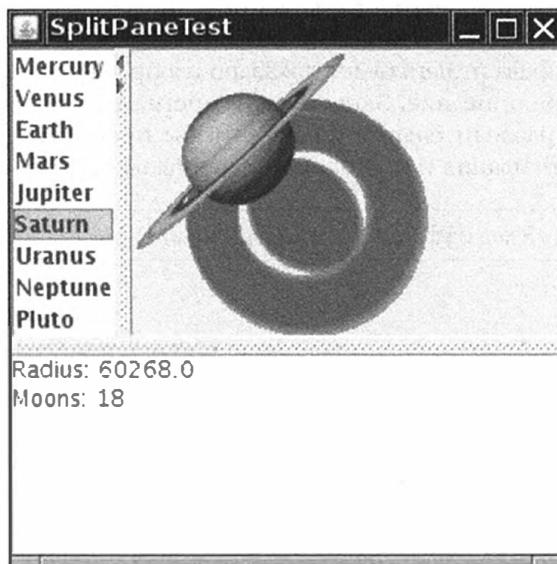


Рис. 10.44. Фрейм с двумя вложенными разделяемыми панелями

При создании разделяемой панели следует указать ориентацию разделятельной полосы по горизонтали (константа `JSplitPane.HORIZONTAL_SPLIT`) или по вертикали (константа `JSplitPane.VERTICAL_SPLIT`), а также два разделяемых компонента, как показано ниже.

```
JSplitPane innerPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,  
    planetList, planetImage);
```

Вот, собственно, и все, что требуется сделать. По желанию разделятельную полосу можно дополнить пиктограммами оперативного развертывания и свертывания панелей, показанными на рис. 10.44. Для визуального стиля Metal они имеют вид треугольников. Достаточно щелкнуть на такой пиктограмме, чтобы разделятельная полоса переместилась в указанном направлении до полного развертывания или свертывания разделяемой панели. Для реализации такой дополнительной возможности достаточно вызвать следующий метод:

```
innerPane.setOneTouchExpandable(true);
```

Кроме того, можно воспользоваться функцией непрерывной компоновки для перерисовки содержимого компонентов разделяемой панели по мере перемещения разделительной полосы. Результаты выглядят очень эффектно, но выполнение кода замедляется. Поэтому данную функцию лучше отключить, как показано ниже.

```
innerPane.setContinuousLayout(true);
```

В рассматриваемом здесь примере программы нижняя разделительная полоса отображается в исходном режиме без непрерывной компоновки. При ее перетаскивании перемещается только черный контур границы раздела. А компоненты разделяемой панели перерисовываются только после отпускания кнопки мыши. В этой простой программе, исходный код которой приведен в листинге 10.29, создается список, заполняемый названиями планет. Если выбрать какую-нибудь планету из списка, то справа от него будет показано изображение выбранной планеты, а снизу — ее краткое описание. Запустив эту программу на выполнение, попробуйте переместить разделительные полосы, чтобы проверить действие функций оперативного развертывания и постоянной перерисовки разделяемых панелей.

Листинг 10.29. Исходный код из файла splitPane/SplitPaneFrame.java

```
1 package splitPane;
2
3 import java.awt.*;
4
5 import javax.swing.*;
6
7 /**
8  * Этот фрейм состоит из двух вложенных разделяемых панелей для
9  * показа изображения и краткого описания выбранной планеты
10 */
11 class SplitPaneFrame extends JFrame
12 {
13     private static final int DEFAULT_WIDTH = 300;
14     private static final int DEFAULT_HEIGHT = 300;
15
16     private Planet[] planets = { new Planet("Mercury", 2440, 0),
17         new Planet("Venus", 6052, 0),
18         new Planet("Earth", 6378, 1),
19         new Planet("Mars", 3397, 2),
20         new Planet("Jupiter", 71492, 16),
21         new Planet("Saturn", 60268, 18),
22         new Planet("Uranus", 25559, 17),
23         new Planet("Neptune", 24766, 8),
24         new Planet("Pluto", 1137, 1), };
25
26     public SplitPaneFrame()
27     {
28         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
29
30         // установить компоненты для названий, изображений
31         // и кратких описаний планет
32
33         final JList<Planet> planetList = new JList<>(planets);
34         final JLabel planetImage = new JLabel();
35         final JTextArea planetDescription = new JTextArea();
```

```

36     planetList.addListSelectionListener(
37         new ListSelectionListener()
38     {
39         public void valueChanged(ListSelectionEvent event)
40         {
41             Planet value = (Planet) planetList.getSelectedValue();
42             // обновить изображение и краткое описание планеты
43             planetImage.setIcon(value.getImage());
44             planetDescription.setText(value.getDescription());
45         }
46     });
47
48     // установить разделяемые панели
49     JSplitPane innerPane = new JSplitPane(
50         JSplitPane.HORIZONTAL_SPLIT, planetList, planetImage);
51
52     innerPane.setContinuousLayout(true);
53     innerPane.setOneTouchExpandable(true);
54
55     JSplitPane outerPane = new JSplitPane(
56         JSplitPane.VERTICAL_SPLIT, innerPane, planetDescription);
57
58     add(outerPane, BorderLayout.CENTER);
59 }
60 }
61 }
62 }
63 }
```

`javax.swing.JSplitPane 1.2`

- **`JSplitPane()`**
- **`JSplitPane(int direction)`**
- **`JSplitPane(int direction, boolean continuousLayout)`**
- **`JSplitPane(int direction, Component first, Component second)`**
- **`JSplitPane(int direction, boolean continuousLayout, Component first, Component second)`**

Конструируют новую разделяемую панель.

Параметры:	<code>direction</code>	Принимает значение одной из следующих констант: HORIZONTAL_SPLIT или VERTICAL_SPLIT
	<code>continuousLayout</code>	Принимает логическое значение true , если содержимое компонентов непрерывно обновляется при перемещении разделяльной полосы
	<code>first, second</code>	Добавляемые компоненты

javax.swing.JSplitPane 1.2 (окончание)

- **boolean isOneTouchExpandable()**
- **void setOneTouchExpandable(boolean b)**

Получают или устанавливают значение свойства, определяющего порядок оперативного развертывания разделяемой панели. Если это свойство принимает логическое значение **true**, на разделительной полосе появляются пиктограммы для развертывания или свертывания одного или другого компонента панели одним щелчком.

- **boolean isContinuousLayout()**
- **void setContinuousLayout(boolean b)**

Получают или устанавливают значение свойства, определяющего режим непрерывной компоновки содержимого разделяемой панели. Если это свойство принимает логическое значение **true**, то при перемещении разделительной полосы компоненты панели непрерывно обновляются.

- **void setLeftComponent(Component c)**
 - **void setTopComponent(Component c)**
- Эти методы дают один и тот же результат, устанавливая в качестве параметра *c* первый компонент на разделяемой панели.
- **void setRightComponent(Component c)**
 - **void setBottomComponent(Component c)**
- Эти методы дают один и тот же результат, устанавливая в качестве параметра *c* второй компонент на разделяемой панели.

10.6.2. Панели с вкладками

Панели с вкладками можно использовать для создания очень удобного пользовательского интерфейса. В этом случае сложное диалоговое окно разбивается на несколько частей, каждая из которых содержит группу связанных по смыслу вариантов выбора. Вкладки можно также применять для просмотра документов или рисунков, как показано на рис. 10.45 и демонстрируется в рассматриваемом здесь примере программы.

Чтобы создать панель с вкладками, необходимо прежде всего сконструировать объект типа **JTabbedPane**, а затем ввести в него вкладки следующим образом:

```
JTabbedPane tabbedPane = new JTabbedPane();
tabbedPane.addTab(title, icon, component);
```

Последний параметр метода **addTab()** относится к типу **Component**. Для ввода нескольких компонентов на одной и той же вкладке необходимо сначала упаковать их в контейнер (например, **JPanel**). Пиктограмму можно и не указывать в качестве параметра *icon*. Для этого предусмотрен другой вариант вызова метода **addTab()**:

```
tabbedPane.addTab(title, component);
```

Очередную вкладку можно ввести в нужном месте уже существующего ряда вкладок. Для этого необходимо указать ее индекс следующим образом:

```
tabbedPane.insertTab(title, icon, component, tooltip, index);
```

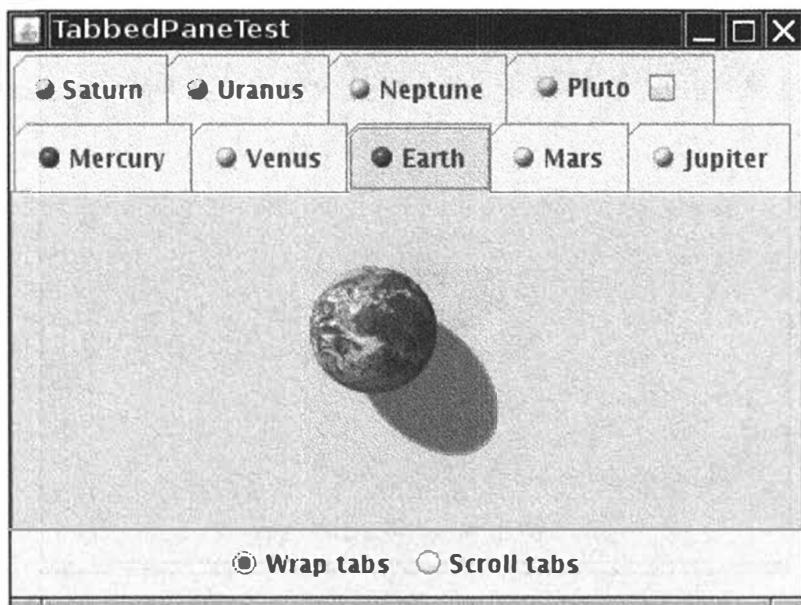


Рис. 10.45. Панель с вкладками

Для удаления ненужной вкладки из ряда вкладок следует вызвать приведенный ниже метод `removeTabAt()`.

```
tabbedPane.removeTabAt(index);
```

Вновь введенная вкладка не отображается автоматически. Для этого ее придется выбрать с помощью метода `setSelectedIndex()`. Например, для отображения вкладки, введенной в конце ряда вкладок, необходимо сделать следующий вызов:

```
tabbedPane.setSelectedIndex(tabbedPane.getTabCount() - 1);
```

Для отображения большого количества вкладок может потребоваться немало места. Ради экономии места в рабочем окне предусмотрена, начиная с версии JDK 1.4, возможность для автоматической прокрутки вкладок, отображаемых в одном ряду. На правом краю от вкладок в этом же ряду присутствуют кнопки со стрелками, обозначающими направление прокрутки (рис. 10.46).

Для указания режима отображения вкладок полностью или частично с возможностью прокрутки вызываются методы

```
tabbedPane.setTabLayoutPolicy(JTabbedPane.WRAP_TAB_LAYOUT);
```

или

```
tabbedPane.setTabLayoutPolicy(JTabbedPane.SCROLL_TAB_LAYOUT);
```

Заголовки вкладок могут иметь мнемонику подобно пунктам меню. Например, в приведенном ниже фрагменте кода прописная буква **M** подчеркивается, и пользователи могут выбрать вкладку, нажав комбинацию клавиш **<Alt+M>**.

```
int marsIndex = tabbedPane.indexOfTab("Mars");
tabbedPane.setMnemonicAt(marsIndex, KeyEvent.VK_M);
```

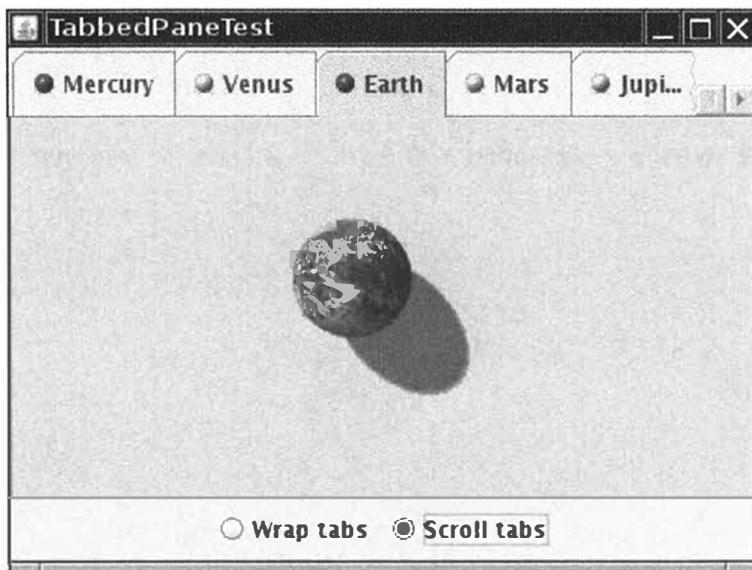


Рис. 10.46. Панель с прокручиваемыми вкладками

В заголовки вкладок можно включать произвольные компоненты. Для этого необходимо сначала ввести вкладку на панели, а затем вызвать следующий метод:

```
tabbedPane.setTabComponentAt(index, component);
```

В рассматриваемом здесь примере программы на вкладке Pluto вводится элемент закрытия, поскольку некоторые астрономы не считают Плутон настоящей планетой. Для этого в качестве компонента данной вкладки устанавливается панель со следующими двумя компонентами: меткой со значком и текстом вкладки, а также флагок с приемником действия, удаляющего вкладку.

В данной программе на примере панели с вкладками демонстрируется следующий удобный прием. Иногда содержимое компонента требуется обновить перед его отображением. Поэтому в данной программе изображение планеты загружается только после того, как пользователь щелкнет кнопкой мыши на соответствующей вкладке. Для уведомления о таком событии необходимо создать обработчик типа *ChangeListener* и связать его с панелью, как показано ниже. Следует, однако, иметь в виду, что такой обработчик создается для панели в целом, а не для отдельного ее компонента.

```
tabbedPane.addChangeListener(listener);
```

Как только пользователь выберет вкладку, вызывается метод *stateChanged()* из класса *ChangeListener*. В качестве источника событий можно выбрать панель с вкладками. Для этого следует вызвать метод *getSelectedIndex()* и определить панель, которую требуется отобразить:

```
public void stateChanged(ChangeEvent event)
{
    int n = tabbedPane.getSelectedIndex();
    loadTab(n);
}
```

В примере программы из листинга 10.30 для всех компонентов вкладки сначала задается пустое значение null. При выборе новой вкладки выполняется проверка компонента на равенство пустому значению null, и при положительном результате компонент заменяется изображением. (Это происходит мгновенно и незаметно для пользователя, который даже не успеет разглядеть пустую панель после того, как щелкнет на вкладке.) Кроме того, ради большей привлекательности цвет шариков в заголовках вкладок меняется с желтого на красный, обозначая те панели, которые уже просматривались.

Листинг 10.30. Исходный код из файла tabbedPane/TabbedPaneFrame.java

```
1 package tabbedPane;
2
3 import java.awt.*;
4
5 import javax.swing.*;
6
7 /**
8  * Этот фрейм содержит панель с вкладками и
9  * кнопки-переключатели режимов отображения
10 * вкладок полностью или частично с прокруткой
11 */
12
13 public class TabbedPaneFrame extends JFrame
14 {
15     private static final int DEFAULT_WIDTH = 400;
16     private static final int DEFAULT_HEIGHT = 300;
17
18     private JTabbedPane tabbedPane;
19
20     public TabbedPaneFrame()
21     {
22         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
23
24         tabbedPane = new JTabbedPane();
25         // установить пустое значение null во всех компонентах и
26         // отложить их загрузку вплоть до показа вкладки в первый раз
27
28         ImageIcon icon = new ImageIcon(
29             getClass().getResource("yellow-ball.gif"));
30
31         tabbedPane.addTab("Mercury", icon, null);
32         tabbedPane.addTab("Venus", icon, null);
33         tabbedPane.addTab("Earth", icon, null);
34         tabbedPane.addTab("Mars", icon, null);
35         tabbedPane.addTab("Jupiter", icon, null);
36         tabbedPane.addTab("Saturn", icon, null);
37         tabbedPane.addTab("Uranus", icon, null);
38         tabbedPane.addTab("Neptune", icon, null);
39         tabbedPane.addTab("Pluto", null, null);
40
41         final int plutoIndex = tabbedPane.indexOfTab("Pluto");
42         JPanel plutoPanel = new JPanel();
43         plutoPanel.add(new JLabel("Pluto", icon,
44             SwingConstants.LEADING));
45         JToggleButton plutoCheckBox = new JCheckBox();
46         plutoCheckBox.addActionListener(event ->
```

```
47                     tabbedPane.remove(plutoIndex));
48         plutoPanel.add(plutoCheckBox);
49         tabbedPane.setTabComponentAt(plutoIndex, plutoPanel);
50
51         add(tabbedPane, "Center");
52
53         tabbedPane.addChangeListener(event ->
54         {
55             // проверить, находится ли пустой компонент
56             // по-прежнему на вкладке
57
58             if (tabbedPane.getSelectedComponent() == null)
59             {
59                 // установить пиктограмму для компонента
60
61                 int n = tabbedPane.getSelectedIndex();
62                 loadTab(n);
63             }
64         });
65     );
66
67     loadTab(0);
68
69     JPanel buttonPanel = new JPanel();
70     ButtonGroup buttonGroup = new ButtonGroup();
71     JRadioButton wrapButton = new JRadioButton("Wrap tabs");
72     wrapButton.addActionListener(event ->
73         tabbedPane.setTabLayoutPolicy(
74             JTabbedPane.WRAP_TAB_LAYOUT));
75     buttonPanel.add(wrapButton);
76     buttonGroup.add(wrapButton);
77     wrapButton.setSelected(true);
78     JRadioButton scrollButton = new JRadioButton("Scroll tabs");
79     scrollButton.addActionListener(event ->
80         tabbedPane.setTabLayoutPolicy(
81             JTabbedPane.SCROLL_TAB_LAYOUT));
82     buttonPanel.add(scrollButton);
83     buttonGroup.add(scrollButton);
84     add(buttonPanel, BorderLayout.SOUTH);
85 }
86
87 /**
88 * Загружает вкладку по указанному индексу
89 * @param n Индекс загружаемой вкладки
90 */
91 private void loadTab(int n)
92 {
93     String title = tabbedPane.getTitleAt(n);
94     ImageIcon planetIcon = new ImageIcon(getClass()
95         .getResource(title + ".gif"));
96     tabbedPane.setComponentAt(n, new JLabel(planetIcon));
97
98     // обозначить данную вкладку как уже просматривавшуюся
99     // просто ради большей привлекательности
100    tabbedPane.setIconAt(n, new ImageIcon(getClass()
101        .getResource("red-ball.gif")));
102 }
103 }
104 }
```

javax.swing.JTabbedPane 1.2

- **JTabbedPane()**
- **JTabbedPane(int placement)**
Конструируют панель с вкладками.
Параметры: **placement** Принимает значение одной из следующих констант:

SwingConstants.TOP,
SwingConstants.LEFT,
SwingConstants.RIGHT или
SwingConstants.BOTTOM
- **void addTab(String title, Component c)**
- **void addTab(String title, Icon icon, Component c)**
- **void addTab(String title, Icon icon, Component c, String tooltip)**
Добавляют вкладку в конце ряда на панели с вкладками.
- **void insertTab(String title, Icon icon, Component c, String tooltip, int index)**
Вставляет вкладку на место по указанному индексу на панели с вкладками.
- **void removeTabAt(int index)**
Удаляет вкладку по указанному индексу из панели с вкладками.
- **void setSelectedIndex(int index)**
Выбирает вкладку по указанному индексу.
- **int getSelectedIndex()**
Возвращает индекс выбранной вкладки.
- **Component getSelectedComponent()**
Возвращает компонент из выбранной вкладки.
- **String getTitleAt(int index)**
- **void setTitleAt(int index, String title)**
- **Icon getIconAt(int index)**
- **void setIconAt(int index, Icon icon)**
- **Component getComponentAt(int index)**
- **void setComponentAt(int index, Component c)**
Получают или устанавливают заголовок, пиктограмму или компонент вкладки по указанному индексу.
- **int indexOfTab(String title)**
- **int indexOfTab(Icon icon)**
- **int indexOfComponent(Component c)**
Возвращают индекс вкладки с указанным заголовком, пиктограммой или компонентом.
- **int getTabCount()**
Возвращает общее количество вкладок на данной панели с вкладками.

javax.swing.JTabbedPane 1.2 (окончание)

- **int getTabLayoutPolicy()**
- **void setTabLayoutPolicy(int policy) 1.4**
Получают или устанавливают порядок отображения вкладок. Они могут отображаться полностью (константа `JTabbedPane.WRAP_TAB_LAYOUT`) или частично с прокруткой (константа `JTabbedPane.SCROLL_TAB_LAYOUT`).
- **int getMnemonicAt(int index) 1.4**
- **void setMnemonicAt(int index, int mnemonic)**
Получают или устанавливают mnemonicический знак на вкладке по указанному индексу. Знак определяется значением константы `VK_X` из класса `KeyEvent`. Значение `-1` обозначает, `Component getTabComponentAt(int index) 6`
- **void setTabComponentAt(int index, Component c) 6**
Получают или устанавливают компонент, отображающий заголовок вкладки по указанному индексу. Если этот компонент пустой, то воспроизводятся значок и заголовок вкладки, а иначе — только указанный компонент на вкладке.
- **int indexOfTabComponent(Component c) 6**
Возвращает индекс вкладки для ее указанного компонента.
- **void addChangeListener(ChangeListener listener)**
Вводит приемник событий, который уведомляется о выборе пользователем другой вкладки.

10.6.3. Настольные панели и внутренние фреймы

Во многих приложениях предоставляемые данные отображаются в нескольких окнах, находящихся в одном крупном фрейме. При свертывании фрейма все эти окна скрываются. В Windows такой пользовательский интерфейс называется **многодокументным (MDI)**. На рис. 10.47 показано типичное приложение, в котором используется такой интерфейс.

Ранее такой стиль представления ГПИ был широко распространен, но в последнее время он стал менее популярным. Так, во многих браузерах для отображения веб-страниц используется несколько фреймов. Какой же способ лучше? Оба способа имеют свои преимущества и недостатки. Многодокументный интерфейс позволяет справиться с беспорядком, который возникает при наличии большого количества открытых окон. Но для перехода между многими окнами верхнего уровня приходится пользоваться кнопками и оперативными клавишами оконной системы базовой платформы.

10.6.3.1. Отображение внутренних фреймов

В межплатформенной среде Java не предусмотрено полагаться на богатые возможности оконной системы базовой платформы. Поэтому целесообразнее строить ГПИ на основе фреймов. На рис. 10.48 показано приложение Java с тремя внутренними фреймами. Два из них содержат пиктограммы для развертывания, свертывания и закрытия фрейма. А третий фрейм показан в свернутом виде.

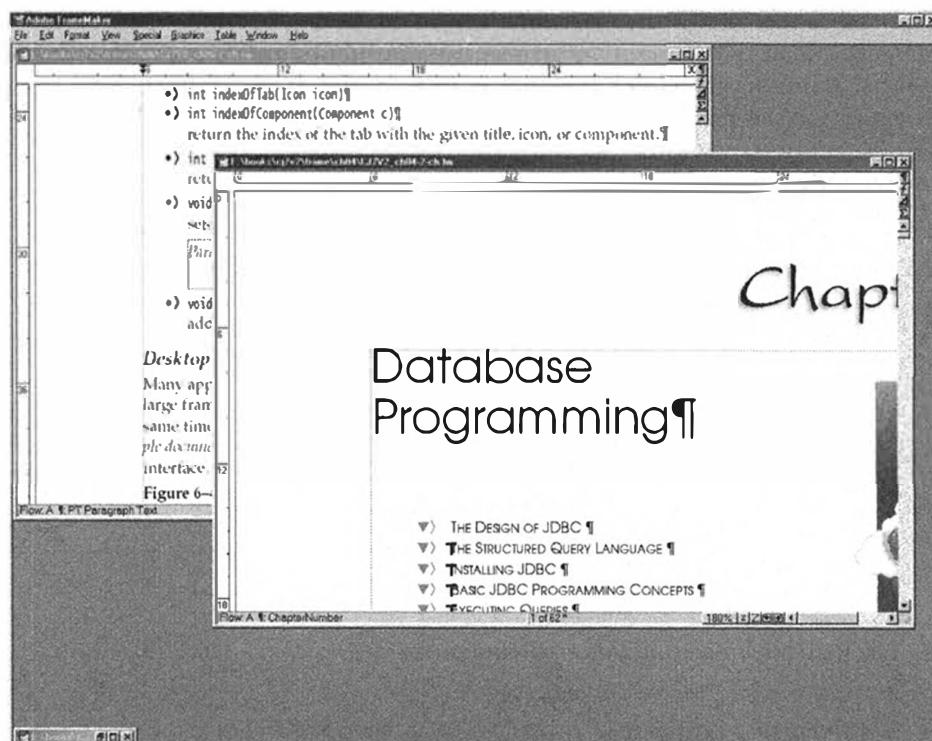


Рис. 10.47. Приложение с многодокументным интерфейсом

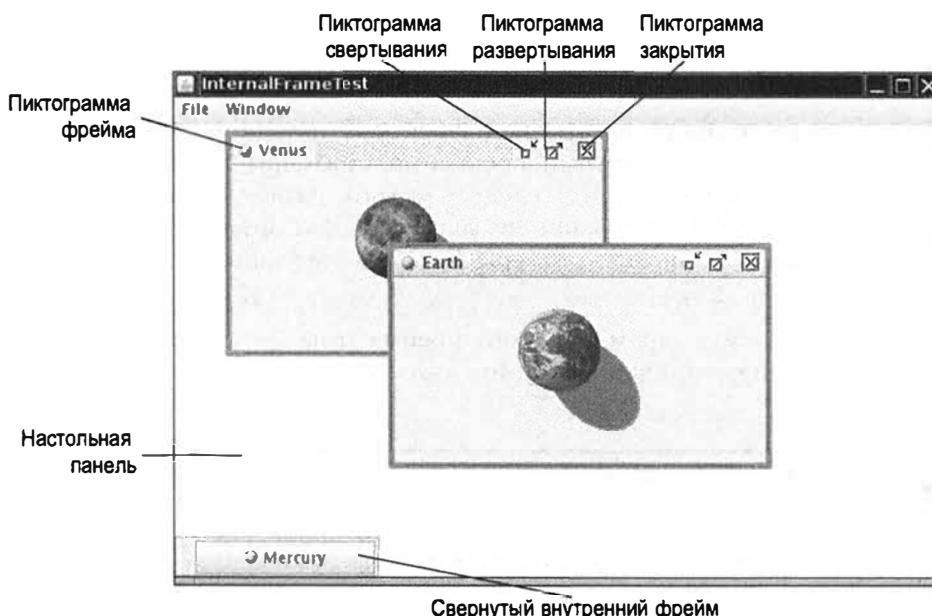


Рис. 10.48. Приложение с тремя внутренними фреймами

При использовании визуального стиля Metal внутренние фреймы имеют отдельные области “захвата”, которые можно использовать для перемещения фреймов. А для изменения размеров окон достаточно перетащить в нужном направлении уголки фрейма.

Чтобы достичь подобных функциональных возможностей фреймов, выполните следующие действия.

1. Создайте обычное окно фрейма типа JFrame для своего приложения.
2. Введите во фрейм типа JFrame настольную панель типа JDesktopPane следующим образом:

```
desktop = new JDesktopPane();
add(desktop, BorderLayout.CENTER);
```

3. Создайте окна внутренних фреймов типа JInternalFrame, указав по желанию пиктограммы для изменения размеров фрейма и его закрытия, как показано ниже. Как правило, указываются все пиктограммы.

```
JInternalFrame iframe = new JInternalFrame(title,
    true, // изменение размеров
    true, // закрытие
    true, // развертывание
    true); // свертывание
```

4. Введите во фрейм нужные компоненты:

```
iframe.add(c, BorderLayout.CENTER);
```

5. Установите для фрейма пиктограмму, которая будет отображаться в его левом верхнем углу:

```
iframe setFrameIcon(icon);
```



На заметку! В текущей версии визуального стиля Metal пиктограмма не отображается, когда фрейм находится в свернутом виде.

6. Установите размеры внутреннего фрейма. В исходном состоянии как внутренний, так и стандартный фрейм имеет размеры 0×0. Кроме того, с помощью метода reshape() следует указать разное исходное положение для всех фреймов, чтобы они не закрывали друг друга. Этот же метод дает возможность установить размеры фреймов следующим образом:

```
iframe.reshape(nextFrameX, nextFrameY, width, height);
```

7. Подобно экземплярам обычного фрейма типа JFrame, сделайте каждый внутренний фрейм видимым, как показано ниже.

```
iframe.setVisible(true);
```



На заметку! В ранних версиях Swing внутренние фреймы автоматически становились видимыми, и поэтому вызывать метод setVisible() не требовалось.

8. Введите фрейм на настольной панели типа JDesktopPane следующим образом:

```
desktop.add(iframe);
```

9. Новый фрейм, вероятнее всего, придется сделать *выбранным фреймом*, который, в отличие от других внутренних фреймов, имеет фокус ввода. В соответствии с визуальным стилем Metal строка заголовка выбранного фрейма выделяется синим цветом, тогда как строка заголовка других невыбранных фреймов — серым. Для выбора фрейма воспользуйтесь методом `setSelected()`. Но свойство, определяющее выбранное состояние фрейма, может оказаться *недоступным для установки*. Это означает, что фрейм, выбранный в настоящий момент, может отказаться отдавать фокус ввода тому фрейму, который требуется выбрать. В таком случае метод `setSelected()` генерирует исключение типа `PropertyVetoException`, которое следует перехватить и обработать:

```
try
{
    iframe.setSelected(true);
}
catch (PropertyVetoException ex)
{
    // в попытке выбрать фрейм было отказано
}
```

10. Расположите каждый последующий фрейм со смещением, чтобы фреймы не закрывали полностью друг друга. Расстояние между фреймами удобно выбрать равным высоте заголовка фрейма, которая получается следующим образом:

```
int frameDistance = iframe.getHeight()
                    - iframe.getContentPane().getHeight()
```

11. Воспользуйтесь полученным расстоянием, чтобы определить положение следующего внутреннего фрейма:

```
nextFrameX += frameDistance;
nextFrameY += frameDistance;
if (nextFrameX + width > desktop.getWidth())
    nextFrameX = 0;
if (nextFrameY + height > desktop.getHeight())
    nextFrameY = 0;
```

10.6.3.2. Каскадное и мозаичное расположение фреймов

В Windows предусмотрено несколько команд для *каскадного* расположения фреймов с перекрытием (рис. 10.49) и *мозаичного* их расположения без перекрытия (рис. 10.50). Но в классах `JDesktopPane` и `JInternalFrame` из библиотеки `Swing` не предусмотрено никаких средств для поддержки подобных операций. Поэтому в примере программы из листинга 10.31 демонстрируется, как самостоятельно организовать упорядочение фреймов, реализовав операции их каскадного и мозаичного расположения.

Для каскадного расположения следует задать одинаковые размеры всех фреймов, сместив положение каждого последующего фрейма на одинаковую величину относительно предыдущего. Для получения массива всех внутренних фреймов вызывается метод `getAllFrames()` из класса `JDesktopPane`:

```
JInternalFrame[] frames = desktop.getAllFrames();
```

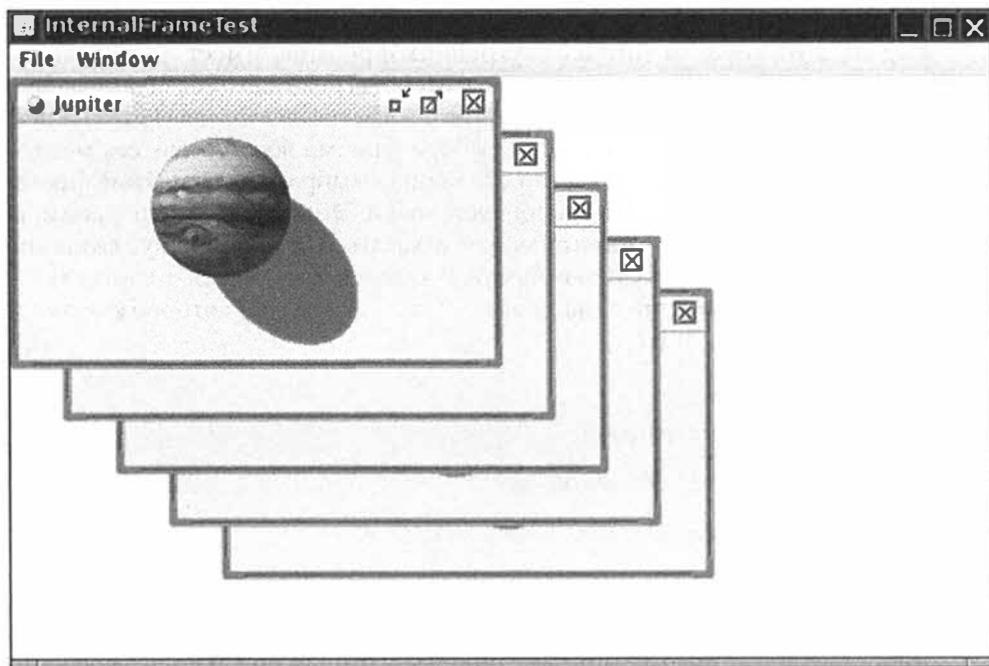


Рис. 10.49. Каскадное расположение внутренних фреймов

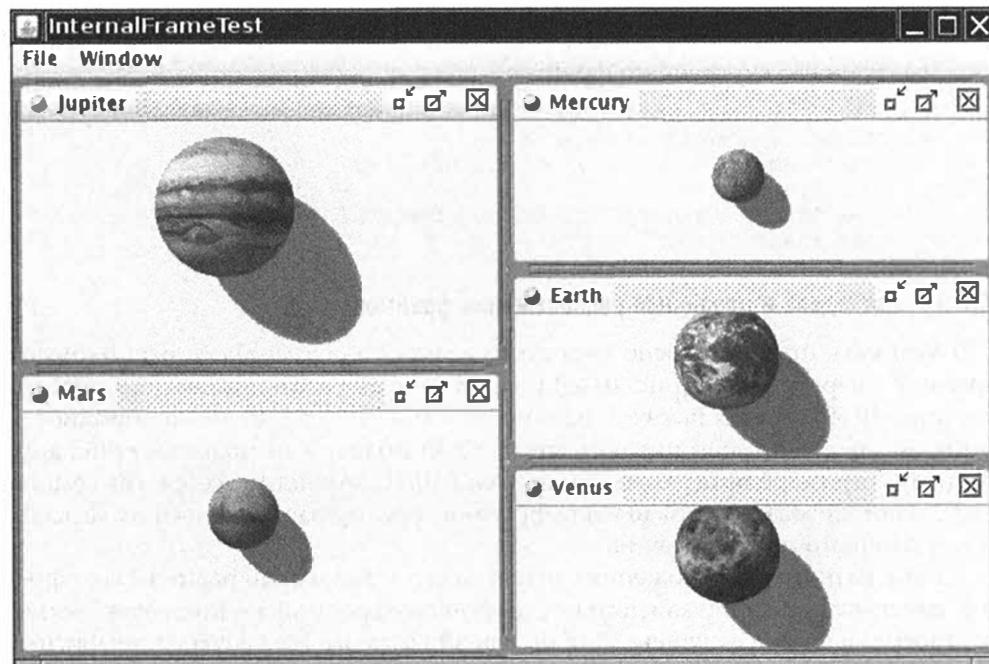


Рис. 10.50. Мозаичное расположение внутренних фреймов

Но при этом следует учитывать текущее состояние внутреннего фрейма. Каждый внутренний фрейм может находиться в одном из следующих трех состояний:

- свернутое;
- нормальное, допускающее изменение размеров;
- развернутое.

Для выявления свернутых фреймов, которые следует пропустить, служит метод `isIcon()`. Но если фрейм развернут, то его нужно сначала установить в нормальное состояние, допускающее изменение размеров, вызвав метод `setMaximum(false)`. Но и это свойство может оказаться недоступным для установки, а следовательно, придется организовать перехват и обработку исключения типа `PropertyVetoException`. В приведенном ниже цикле все внутренние фреймы располагаются каскадом на настольной панели.

```
for (JInternalFrame frame : desktop.getAllFrames())
{
    if (!frame.isIcon())
    {
        try
        {
            // попытаться перевести фреймы из развернутого состояния
            // в нормальное, допускающее изменение размеров, хотя это
            // может быть и запрещено
            frame.setMaximum(false);
            frame.reshape(x, y, width, height);
            x += frameDistance;
            y += frameDistance;
            // перенести в следующий ряд по достижении правого края
            // настольной панели

            if (x + width > desktop.getWidth()) x = 0;
            if (y + height > desktop.getHeight()) y = 0;
        }
        catch (PropertyVetoException ex)
        {}
    }
}
```

Организовать мозаичное расположение фреймов труднее, особенно если их количество таково, что они не вписываются полностью в квадратную сетку. С этой целью сначала подсчитывается количество несвернутых фреймов, а затем определяется количество рядов в первом столбце квадратной сетки:

```
int rows = (int) Math.sqrt(frameCount);
```

Далее определяется количество заполненных столбцов:

```
int cols = frameCount / rows;
```

А последний незаполненный столбец определяется так:

```
int extra = frameCount % rows
```

Количество рядов в этом столбце равно `rows + 1`. В приведенном ниже цикле все внутренние фреймы располагаются мозаикой на настольной панели.

```
int width = desktop.getWidth() / cols;
int height = desktop.getHeight() / rows;
```

```
int r = 0;
int c = 0;
for (JInternalFrame frame : desktop.getAllFrames())
{
    if (!frame.isIcon())
    {
        try
        {
            frame.setMaximum(false);
            frame.reshape(c * width, r * height, width, height);
            r++;
            if (r == rows)
            {
                r = 0;
                c++;
                if (c == cols - extra)
                {
                    // начать ввод дополнительного ряда
                    rows++;
                    height = desktop.getHeight() / rows;
                }
            }
        }
        catch (PropertyVetoException ex)
        {}
    }
}
```

В рассматриваемом здесь примере программы демонстрируется еще одна распространенная операция: перемещение выбора от текущего фрейма к следующему несвернутому фрейму. С этой целью обходятся все фреймы и вызывается метод `isSelected()` до тех пор, пока не будет найден текущий выбранный фрейм. Затем выявляется следующий несвернутый фрейм и делается попытка выбрать его с помощью приведенного ниже метода.

```
rames[next].setSelected(true);
```

Как и прежде, этот метод может генерировать исключение типа `PropertyVetoException`. Возврат к исходному фрейму означает, что других доступных для выбора фреймов не существует. Ниже приведен цикл для обхода всех внутренних фреймов, выявления среди них текущего выбранного фрейма и выбора следующего.

```
JInternalFrame[] frames = desktop.getAllFrames();
for (int i = 0; i < frames.length; i++)
{
    if (frames[i].isSelected())
    {
        // найти следующий несвернутый фрейм, который можно выбрать
        int next = (i + 1) % frames.length;
        while (next != i)
        {
            if (!frames[next].isIcon())
            {
                try
                {
                    // все остальные фреймы свернуты или недоступны для выбора
                    frames[next].setSelected(true);
                }
                catch (Exception e)
                {
                    e.printStackTrace();
                }
            }
            else
            {
                next = (next + 1) % frames.length;
            }
        }
    }
}
```

```
        frames[next].toFront();
        frames[i].toBack();
        return;
    }
    catch (PropertyVetoException ex)
    {}
}
next = (next + 1) % frames.length;
}
}
}
```

10.6.3.3. Наложение запрета на установку свойств

Итак, рассмотрев исключения, возникающие в связи с запретом на установку свойств, выясним, каким образом подобный запрет накладывается во фреймах. Для контроля над установкой свойств в классе `JInternalFrame` используется общий механизм JavaBeans. Этот механизм здесь подробно не рассматривается, но все же показывается, каким образом во фреймах отвергаются запросы на изменение свойств.

Запреты обычно не накладываются на свертывание или передачу фокуса ввода, но нередко во фреймах проверяется, разрешено ли их закрывать. Для закрытия фрейма служит метод `setClosed()` из класса `JInternalFrame`. Но поскольку этот метод допускает наложение запрета, то в нем вызываются все зарегистрированные приемники запрещаемых изменений перед тем, как перейти непосредственно к изменениям. Таким образом, каждому приемнику предоставляется возможность генерировать исключение типа `PropertyVetoException` и завершить метод `setClosed()` до внесения в нем каких-нибудь изменений в установки свойств.

В рассматриваемом здесь примере программы появляется диалоговое окно, приведенное на рис. 10.51. В нем пользователю предлагается подтвердить намерение закрыть фрейм. Если пользователь не дает такого подтверждения, фрейм остается открытым.

Для организации такого уведомления пользователя выполните следующие действия.

1. Создайте объект приемника для каждого фрейма, как показано ниже. Этот объект должен быть экземпляром некоторого класса, реализующего интерфейс `VetoableChangeListener`. Такой приемник лучше всего ввести сразу же после создания фрейма. В рассматриваемом здесь примере программы для создания внутренних фреймов применяется класс фрейма, хотя для этой цели вполне подойдет и анонимный внутренний класс.

```
iframe.addVetoableChangeListener(listener);
```

2. Реализуйте метод `vetoableChange()`, объявленный единственным в интерфейсе `VetoableChangeListener`. В качестве параметра ему передается объект типа `PropertyChangeEvent`. Чтобы получить имя изменяемого свойства, воспользуйтесь методом `getName()`. Так, если для наложения запрета вызывается метод `setClosed(true)`, изменяемое свойство будет называться `closed`. Имя свойства получается путем удаления префикса `set` из имени метода и замены первой прописной буквы на строчную.

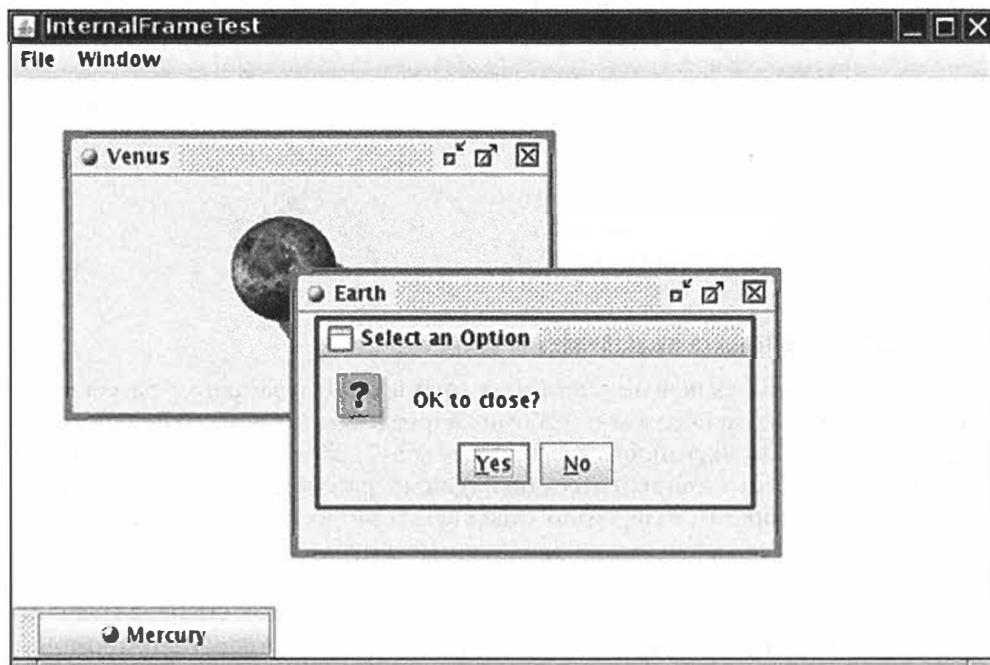


Рис. 10.51. Пользователь может наложить запрет на закрытие окна

3. Вызовите метод `getNewValue()`, чтобы получить новое предлагаемое значение свойства:

```
String name = event.getPropertyName();
Object value = event.getNewValue();
if (name.equals("closed") && value.equals(true))
{
    запросить у пользователя подтверждение
}
```

4. Создайте метод, генерирующий исключение типа `PropertyVetoException`, чтобы наложить запрет на изменение свойства. Если же накладывать запрет на изменение свойства не требуется, завершите метод обычным способом, как показано в приведенном ниже фрагменте кода.

```
class DesktopFrame extends JFrame
    implements VetoableChangeListener
{
    . . .
    public void vetoableChange(PropertyChangeEvent event)
        throws PropertyVetoException
    {
        . . .
        if (запрещено)
            throw new PropertyVetoException(причина_запрета, event);
        // если разрешено, произвести возврат обычным способом
    }
}
```

10.6.3.4. Диалоговые окна во внутренних фреймах

Для создания диалоговых окон во внутренних фреймах не рекомендуется использовать класс `JDialog`, поскольку такие окна обладают следующими недостатками.

- Они весьма ресурсоемки, потому что создают новый фрейм в системе окон.
- Системе управления окнами неизвестно, как расположить диалоговое окно по отношению к породившему его внутреннему фрейму.

Вместо этого для создания простых диалоговых окон рекомендуется пользоваться методами типа `showInternalXxxDialog()` из класса `JOptionPane`. Они действуют аналогично методам типа `showXxxDialog()`, за исключением того, что диалоговое окно располагается непосредственно над внутренним фреймом. Для создания более сложных диалоговых окон следует воспользоваться классом `JInternalFrame`, хотя в нем не предусмотрены инструменты для поддержки модельных (т.е. режимных) диалоговых окон.

В рассматриваемом здесь примере программы используется внутреннее диалоговое окно, в котором пользователю предлагается подтвердить закрытие фрейма, как выделено ниже полужирным.

```
int result = JOptionPane.showInternalConfirmDialog(
    iframe, "OK to close?", "Select an Option",
    JOptionPane.YES_NO_OPTION);
```



На заметку! Если требуется только уведомить о закрытии фрейма, вместо механизма запрета лучше установить приемник типа `InternalFrameListener`, который действует аналогично приемнику типа `WindowsListener`. При закрытии внутреннего фрейма вместо метода `windowClosing()` вызывается метод `internalFrameClosing()`. Аналогичным образом применяются все шесть методов, уведомляющих о действиях с внутренним фреймом [открыт/закрыт, свернут/развернут, активен/неактивен].

10.6.3.5. Перетаскивание контуров

Разработчики часто жалуются на низкую производительность при использовании внутренних фреймов. Например, медленнее всего выполняется операция перетаскивания фрейма со сложным содержимым. Дело в том, что настольный диспетчер постоянно запрашивает повторное воспроизведение фрейма, что существенно замедляет дело.

Подобная ситуация наблюдается в графических оболочках `Windows` или `X Window` с неудачно написанным видеодрайвером. В некоторых системах перетаскивание выполняется быстро, поскольку в видеоадаптерах данная операция поддерживается на аппаратном уровне. С этой целью изображение во фрейме отображается в разные места экрана в процессе перетаскивания.

Для повышения производительности можно воспользоваться так называемым *перетаскиванием контуров*. В этом случае при перетаскивании фрейма постоянно обновляются только его контуры. А внутренняя часть фрейма перерисовывается только тогда, когда фрейм достигнет своего конечного положения. Чтобы включить режим перетаскивания контуров, следует вызвать приведенный ниже метод.

Такая установка равнозначна переключению режима постоянной компоновки в классе `JSplitPane`.

```
desktop.setDragMode(JDesktopPane.OUTLINE_DRAG_MODE);
```



На заметку! В ранних версиях библиотеки Swing для включения режима перетаскивания контуров приходилось вызывать следующий метод:

```
desktop.putClientProperty("JDesktopPane.dragMode", "outline");
```

В рассматриваемом здесь примере программы для включения/отключения режима перетаскивания контуров пользователю предоставляется пункт меню `Window⇒Drag Outline` (Окно⇒Перетащить контур) с устанавливаемым флажком.



На заметку! Внутренними фреймами на настольной панели управляет настольный диспетчер, реализуемый в классе `DesktopManager`. Для обычного программирования этот класс не требуется. Его следует применять лишь для реализации различных вариантов поведения настольной панели. С этой целью устанавливается новый настольный диспетчер, но рассмотрение подобных вопросов выходит за рамки данной книги.

В рассматриваемом здесь примере программы из листинга 10.31 демонстрируется настольная панель с внутренними фреймами, предназначенными для отображения HTML-страниц. При выборе пункта меню `File⇒Open` (Файл⇒Открыть) на экране появляется диалоговое окно для выбора локального HTML-файла. Если щелкнуть на выбранном HTML-файле, он отобразится в новом внутреннем фрейме. Попробуйте изменить расположение фреймов на экране с помощью пунктов меню `Window⇒Cascade` (Окно⇒Расположить каскадом) и `Window⇒Tile` (Окно⇒Расположить мозаикой).

Листинг 10.31. Исходный код из файла `internalFrame/DesktopFrame.java`

```
1 package internalFrame;
2
3 import java.awt.*;
4 import java.beans.*;
5
6 import javax.swing.*;
7
8 /**
9  * Этот настольный фрейм содержит панели редакторов,
10 * на которых отображаются HTML-документы
11 */
12 public class DesktopFrame extends JFrame
13 {
14     private static final int DEFAULT_WIDTH = 600;
15     private static final int DEFAULT_HEIGHT = 400;
16     private static final String[] planets =
17         { "Mercury", "Venus", "Earth", "Mars", "Jupiter",
18           "Saturn", "Uranus", "Neptune", "Pluto", };
19
20     private JDesktopPane desktop;
21     private int nextFrameX;
22     private int nextFrameY;
23     private int frameDistance;
```

```
24     private int counter;
25
26     public DesktopFrame()
27     {
28         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
29
30         desktop = new JDesktopPane();
31         add(desktop, BorderLayout.CENTER);
32
33         // установить меню
34
35         JMenuBar menuBar = new JMenuBar();
36         setJMenuBar(menuBar);
37         JMenu fileMenu = new JMenu("File");
38         menuBar.add(fileMenu);
39         JMenuItem openItem = new JMenuItem("New");
40         openItem.addActionListener(event ->
41             {
42                 createInternalFrame(new JLabel(new ImageIcon(getClass()
43                     .getResource(planets[counter] + ".gif"))),
44                     planets[counter]);
45                 counter = (counter + 1) % planets.length;
46             });
47         fileMenu.add(openItem);
48         JMenuItem exitItem = new JMenuItem("Exit");
49         exitItem.addActionListener(event -> System.exit(0));
50         fileMenu.add(exitItem);
51         JMenu windowMenu = new JMenu("Window");
52         menuBar.add(windowMenu);
53         JMenuItem nextItem = new JMenuItem("Next");
54         nextItem.addActionListener(event -> selectNextWindow());
55         windowMenu.add(nextItem);
56         JMenuItem cascadeItem = new JMenuItem("Cascade");
57         cascadeItem.addActionListener(event -> cascadeWindows());
58         windowMenu.add(cascadeItem);
59         JMenuItem tileItem = new JMenuItem("Tile");
60         tileItem.addActionListener(event -> tileWindows());
61         windowMenu.add(tileItem);
62         final JCheckBoxMenuItem dragOutlineItem =
63             new JCheckBoxMenuItem("Drag Outline");
64         dragOutlineItem.addActionListener(event ->
65             desktop.setDragMode(dragOutlineItem.isSelected()
66                 ? JDesktopPane.OUTLINE_DRAG_MODE
67                 : JDesktopPane.LIVE_DRAG_MODE));
68         windowMenu.add(dragOutlineItem);
69     }
70
71 /**
72 * Создает внутренний фрейм на настольной панели
73 * @param c Компонент, отображаемый во внутреннем фрейме
74 * @param t Заголовок внутреннего фрейма
75 */
76 public void createInternalFrame(Component c, String t)
77 {
78     final JInternalFrame iframe = new JInternalFrame(t,
79             true, // изменение размеров
80             true, // закрытие
81             true, // разворачивание
```

```
82                     true); // сворачивание
83
84         iframe.add(c, BorderLayout.CENTER);
85         desktop.add(iframe);
86
87         iframe.setFrameIcon(new ImageIcon(
88             getClass().getResource("document.gif")));
89
90         // ввести приемник, чтобы подтвердить закрытие фрейма
91         iframe.addVetoableChangeListener(event ->
92         {
93             String name = event.getPropertyName();
94             Object value = event.getNewValue();
95
96             // проверить только попытки закрыть фрейм
97             if (name.equals("closed") && value.equals(true))
98             {
99                 // запросить у пользователя разрешение закрыть фрейм
100                int result = JOptionPane.showInternalConfirmDialog(
101                                iframe, "OK to close?",
102                                "Select an Option",
103                                JOptionPane.YES_NO_OPTION);
104
105                // если пользователь не дает разрешение,
106                // наложить запрет на закрытие фрейма
107                if (result != JOptionPane.YES_OPTION)
108                    throw new PropertyVetoException(
109                        "User canceled close", event);
110            }
111        });
112
113        // расположить фрейм
114        int width = desktop.getWidth() / 2;
115        int height = desktop.getHeight() / 2;
116        iframe.reshape(nextFrameX, nextFrameY, width, height);
117
118        iframe.show();
119
120        // выбрать фрейм, на что может быть наложен запрет
121        try
122        {
123            iframe.setSelected(true);
124        }
125        catch (PropertyVetoException ex)
126        {
127        }
128
129        frameDistance = iframe.getHeight()
130                  - iframe.getContentPane().getHeight();
131
132        // вычислить расположение следующего фрейма
133
134        nextFrameX += frameDistance;
135        nextFrameY += frameDistance;
136        if (nextFrameX + width >
137            desktop.getWidth()) nextFrameX = 0;
138        if (nextFrameY + height >
139            desktop.getHeight()) nextFrameY = 0;
```

```
140    }
141
142    /**
143     * Располагает каскадом внутренние фреймы,
144     * не свернутые на настольной панели
145     */
146    public void cascadeWindows()
147    {
148        int x = 0;
149        int y = 0;
150        int width = desktop.getWidth() / 2;
151        int height = desktop.getHeight() / 2;
152
153        for (JInternalFrame frame : desktop.getAllFrames())
154        {
155            if (!frame.isIcon())
156            {
157                try
158                {
159                    // попытаться сделать развернутые фреймы допускающими
160                    // изменение размеров, что может быть запрещено
161                    frame.setMaximum(false);
162                    frame.reshape(x, y, width, height);
163
164                    x += frameDistance;
165                    y += frameDistance;
166                    // перенести в следующий ряд по достижении правого
167                    // края настольной панели
168                    if (x + width > desktop.getWidth()) x = 0;
169                    if (y + height > desktop.getHeight()) y = 0;
170                }
171                catch (PropertyVetoException ex)
172                {
173                }
174            }
175        }
176    }
177
178    /**
179     * Располагает мозаикой несвернутые внутренние фреймы
180     * на настольной панели
181     */
182    public void tileWindows()
183    {
184        // подсчитать количество несвернутых фреймов
185        int frameCount = 0;
186        for (JInternalFrame frame : desktop.getAllFrames())
187            if (!frame.isIcon()) frameCount++;
188        if (frameCount == 0) return;
189
190        int rows = (int) Math.sqrt(frameCount);
191        int cols = frameCount / rows;
192        int extra = frameCount % rows;
193        // количество столбцов с дополнительными рядами
194
195        int width = desktop.getWidth() / cols;
196        int height = desktop.getHeight() / rows;
197        int r = 0;
```

```
198     int c = 0;
199     for (JInternalFrame frame : desktop.getAllFrames())
200     {
201         if (!frame.isIcon())
202         {
203             try
204             {
205                 frame.setMaximum(false);
206                 frame.reshape(c * width, r * height, width, height);
207                 r++;
208                 if (r == rows)
209                 {
210                     r = 0;
211                     c++;
212                     if (c == cols - extra)
213                     {
214                         // начать ввод дополнительного ряда
215                         rows++;
216                         height = desktop.getHeight() / rows;
217                     }
218                 }
219             }
220             catch (PropertyVetoException ex)
221             {
222             }
223         }
224     }
225 }
226
227 /**
228 * Перемещает вперед следующий несвернутый внутренний фрейм
229 */
230 public void selectNextWindow()
231 {
232     JInternalFrame[] frames = desktop.getAllFrames();
233     for (int i = 0; i < frames.length; i++)
234     {
235         if (frames[i].isSelected())
236         {
237             // найти следующий несвернутый фрейм,
238             // который можно выбрать
239             int next = (i + 1) % frames.length;
240             while (next != i)
241             {
242                 if (!frames[next].isIcon())
243                 {
244                     try
245                     {
246                         // все остальные фреймы свернуты
247                         // или их запрещено выбирать
248                         frames[next]. setSelected(true);
249                         frames[next].toFront();
250                         frames[i].toBack();
251                         return;
252                     }
253                     catch (PropertyVetoException ex)
254                     {
255                     }
256                 }
257             }
258         }
259     }
260 }
```

```

256         }
257         next = (next + 1) % frames.length;
258     }
259 }
260 }
261 }
262 }

```

javax.swing.JDesktopPane 1.2

- **JInternalFrame[] getAllFrames()**

Получает все внутренние фреймы на данной панели.

- **void setDragMode(int mode)**

Устанавливает на время перетаскивания фрейма отображение его содержимого или только контуров.

Параметры: **mode** Принимает значение одной из следующих констант:

JDesktopPane.LIVE_DRAG_MODE или

JDesktopPane.OUTLINE_DRAG_MODE

javax.swing.JInternalFrame 1.2

- **JInternalFrame()**

- **JInternalFrame(String title)**

- **JInternalFrame(String title, boolean resizable)**

- **JInternalFrame(String title, boolean resizable, boolean closable)**

- **JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable)**

- **JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable, boolean iconifiable)**

Конструируют новый внутренний фрейм.

Параметры: **title**

Символьная строка для вывода

заголовка внутреннего фрейма

resizable

Принимает логическое значение

true, разрешающее изменение
размеров внутреннего фрейма

closable

Принимает логическое значение

true, разрешающее закрытие
внутреннего фрейма

maximizable

Принимает логическое значение

true, разрешающее разворачивание
внутреннего фрейма

iconifiable

Принимает логическое значение

true, разрешающее сворачивание
внутреннего фрейма

javax.swing.JInternalFrame 1.2 (продолжение)

- **boolean isResizable()**
- **void setResizable(boolean b)**
- **boolean isClosable()**
- **void setClosable(boolean b)**
- **boolean isMaximizable() void setMaximizable(boolean b)**
- **boolean isIconifiable()**
- **void setIconifiable(boolean b)**

Получают или устанавливают свойства **resizable**, **closable**, **maximizable** или **iconifiable**. Если соответствующее свойство принимает логическое значение **true**, в строке заголовка фрейма отображается пиктограмма изменения размеров, закрытия, разворачивания или сворачивания внутреннего фрейма.

- **boolean isIcon()**
- **void setIcon(boolean b)**
- **boolean isMaximum()**
- **void setMaximum(boolean b)**
- **boolean isClosed()**
- **void setClosed(boolean b)**

Получают или устанавливают свойства **icon**, **maximum** или **closed**. Если соответствующее свойство принимает логическое значение **true**, то допускается разворачивание, сворачивание или закрытие внутреннего фрейма.

- **boolean isSelected()**
- **void setSelected(boolean b)**

Получают или устанавливают свойство **selected**. Если данное свойство принимает логическое значение **true**, текущий внутренний фрейм становится выбранным на настольной панели.

- **void moveToFront()**
- **void moveToBack()**

Перемещают внутренний фрейм на настольной панели вперед или назад.

- **void reshape(int x, int y, int width, int height)**

Перемещает внутренний фрейм и изменяет его размеры.

Параметры:

x, y

Координаты верхнего левого

внутреннего угла фрейма

width, height

Ширина и высота фрейма

- **Container getContentPane()**
- **void setContentPane(Container c)**

Получают или устанавливают панель содержимого для данного внутреннего фрейма.

- **JDesktopPane getDesktopPane()**

Получает настольную панель для данного внутреннего фрейма.

javax.swing.JInternalFrame 1.2 (окончание)

- **Icon getFrameIcon()**
- **void setFrameIcon(Icon anIcon)**
Получают или устанавливают пиктограмму, которая отображается в строке заголовка внутреннего фрейма.
- **boolean isVisible()**
- **void setVisible(boolean b)**
Получают или устанавливают свойство, определяющее видимость внутреннего фрейма.
- **void show()**
Делает видимым данный внутренний фрейм и перемещает его вперед.

javax.swing.JComponent 1.2

- **void addVetoableChangeListener(VetoableChangeListener listener)**
Вводит приемник запрещаемых изменений, который уведомляется при попытке изменить свойство, если на это наложен запрет.

java.beans.VetoableChangeListener 1.1

- **void vetoableChange(PropertyChangeEvent event)**
Вызывается, когда метод установки свойства, на изменение которого наложен запрет, соответственно уведомляет приемник запрещаемых изменений.

java.beans.PropertyChangeEvent 1.1

- **String getPropertyName()**
Возвращает имя изменяемого свойства.
- **Object getNewValue()**
Возвращает предлагаемое новое значение свойства.

java.beans.PropertyVetoException 1.1

- **PropertyVetoException (String reason, PropertyChangeEvent event)**
Генерирует исключение для свойства, изменения в котором запрещены.
- | | | |
|------------|---------------|--|
| Параметры: | reason | Причина наложения запрета |
| | event | Событие, наступающее при попытке нарушить запрет |

10.6.4. Слои

В версии Java SE 1.7 внедрено средство, позволяющее накладывать компоненты ГПИ слоями. В слое можно рисовать, принимая события из нижележащего слоя. Слои позволяют вводить дополнительные визуальные ориентиры в ГПИ. Например, с их помощью можно специально оформить введенные в настоящий момент данные, неверно введенные данные или недоступные компоненты.

Класс `JLayer` связывает компонент с объектом типа `LayerUI`, отвечающим за воспроизведение и обработку событий. У конструктора класса `LayerUI` имеется один параметр типа, который должен соответствовать связанному компоненту. В качестве примера в приведенном ниже фрагменте кода показано, каким образом компонент типа `JPanel` дополняется новым слоем.

```
 JPanel panel = new JPanel();
 LayerUI<JPanel> layerUI = new PanelLayer();
 JLayer layer = new JLayer(panel, layerUI);
 frame.add(layer);
```

Следует, однако, иметь в виду, что в родительский компонент вводится слой, а не панель. Как показано ниже, класс `PanelLayer` является производным от класса `LayerUI`.

```
 class PanelLayer extends LayerUI<Panel>
{
    public void paint(Graphics g, JComponent c)
    {
        . . .
    }
    . . .
}
```

В методе `paint()` допускается рисовать все, что угодно. Тем не менее для рисования компонента следует сделать вызов `super.paint()`. Так, в приведенном ниже фрагменте кода весь компонент окрашивается прозрачным цветом.

```
public void paint(Graphics g, JComponent c)
{
    super.paint(g, c);
    Graphics2D g2 = (Graphics2D) g.create();
    g2.setComposite(AlphaComposite.getInstance(
        AlphaComposite.SRC_OVER, .3f));
    g2.setPaint(color);
    g2.fillRect(0, 0, c.getWidth(), c.getHeight());
    g2.dispose();
}
```

Для приема событий от связанного компонента или любых производных от него компонентов в классе `LayerUI` должна быть установлена маска событий в слое. Это должно быть сделано в методе `installUI()` следующим образом:

```
 class PanelLayer extends LayerUI<JPanel>
{
    public void installUI(JComponent c)
    {
        super.installUI(c);
        ((JLayer<?>) c).setLayerEventMask(AWTEvent.KEY_EVENT_MASK |
```

```

        AWTEvent.FOCUS_EVENT_MASK);
}

public void uninstallUI(JComponent c)
{
    ((JLayer<?>) c).setLayerEventMask(0);
    super.uninstallUI(c);
}
...
}

```

Теперь события будут приниматься в методах типа `processXxxEvent()`. Так, в рассматриваемом здесь примере программы слой перерисовывается после каждого нажатия клавиш:

```

public class PanelLayer extends LayerUI< JPanel >
{
    protected void processKeyEvent(KeyEvent e, JLayer<? extends JPanel> l)
    {
        l.repaint();
    }
}

```

В рассматриваемом здесь примере программы из листинга 10.32 имеются три поля для ввода значений основных (RGB) составляющих цвета фона. Как только пользователь изменит эти значения, цвет сразу же отображается на панели. Кроме того, в данной программе перехватываются события, связанные с перемещением фокуса ввода, а текст в компоненте, обладающем фокусом ввода, выделяется полужирным шрифтом.

Листинг 10.32. Исходный код из файла layer/ColorFrame.java

```

1 package layer;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6 import javax.swing.plaf.*;
7
8 /**
9  * Фрейм с тремя текстовыми полями для установки цвета фона
10 */
11 public class ColorFrame extends JFrame
12 {
13     private JPanel panel;
14     private JTextField redField;
15     private JTextField greenField;
16     private JTextField blueField;
17
18     public ColorFrame()
19     {
20         panel = new JPanel();
21
22         panel.add(new JLabel("Red:"));
23         redField = new JTextField("255", 3);

```

```
24     panel.add(redField);
25
26     panel.add(new JLabel("Green:"));
27     greenField = new JTextField("255", 3);
28     panel.add(greenField);
29     panel.add(new JLabel("Blue:"));
30     blueField = new JTextField("255", 3);
31     panel.add(blueField);
32
33     LayerUI<JPanel> layerUI = new PanelLayer();
34     JLayer<JPanel> layer = new JLayer<JPanel>(panel, layerUI);
35
36     add(layer);
37     pack();
38 }
39
40 class PanelLayer extends LayerUI<JPanel>
41 {
42     public void installUI(JComponent c)
43     {
44         super.installUI(c);
45         ((JLayer<?>) c).setLayerEventMask(
46             AWTEvent.KEY_EVENT_MASK | AWTEvent.FOCUS_EVENT_MASK);
47     }
48
49     public void uninstallUI(JComponent c)
50     {
51         ((JLayer<?>) c).setLayerEventMask(0);
52         super.uninstallUI(c);
53     }
54
55     protected void processKeyEvent(
56         KeyEvent e, JLayer<? extends JPanel> l)
57     {
58         l.repaint();
59     }
60
61     protected void processFocusEvent(
62         FocusEvent e, JLayer<? extends JPanel> l)
63     {
64         if (e.getID() == FocusEvent.FOCUS_GAINED)
65         {
66             Component c = e.getComponent();
67             c.setFont(getFont().deriveFont(Font.BOLD));
68         }
69         if (e.getID() == FocusEvent.FOCUS_LOST)
70         {
71             Component c = e.getComponent();
72             c.setFont(getFont().deriveFont(Font.PLAIN));
73         }
74     }
75
76     public void paint(Graphics g, JComponent c)
77     {
78         super.paint(g, c);
79         Graphics2D g2 = (Graphics2D) g.create();
80         g2.setComposite(AlphaComposite.getInstance(
81             AlphaComposite.SRC_IN));
```

```

82         AlphaComposite.SRC_OVER, .3f));
83     int red = Integer.parseInt(redField.getText().trim());
84     int green = Integer.parseInt(greenField.getText().trim());
85     int blue = Integer.parseInt(blueField.getText().trim());
86     g2.setPaint(new Color(red, green, blue));
87     g2.fillRect(0, 0, c.getWidth(), c.getHeight());
88     g2.dispose();
89   }
90 }
91 }
```

javax.swing.JLayer<V extends Component> 7

- **JLayer(V view, LayerUI<V> ui)**

Создает слой над заданным представлением **view**, делегируя воспроизведение и обработку событий указанному объекту **ui**.

- **void setLayerEventMask(long layerEventMask)**

Позволяет отправить объекту типа **LayerUI** все совпадающие события, посланные связанным с ним компоненту или производным от него компонентам. Для составления маски событий в слое применяются в любом сочетании следующие константы из класса **AWTEvent**:

COMPONENT_EVENT_MASK
FOCUS_EVENT_MASK
HIERARCHY_BOUNDS_EVENT_MASK
HIERARCHY_EVENT_MASK
INPUT_METHOD_EVENT_MASK
KEY_EVENT_MASK
MOUSE_EVENT_MASK
MOUSE_MOTION_EVENT_MASK
MOUSE_WHEEL_EVENT_MASK

javax.swing.plaf.LayerUI<V extends Component> 7

- **void installUI(JComponent c)**
- **void uninstallUI(JComponent c)**

Вызываются при установке или удалении объекта типа **LayerUI**, связанного с заданным компонентом **c**. Этот метод следует переопределить для установки или удаления маски событий в слое.

- **void paint(Graphics g, JComponent c)**

Вызывается при рисовании специально оформленяемого компонента. Этот метод следует переопределить для вызова метода **super.paint()** и воспроизведения специального оформления.

- **void processComponentEvent(ComponentEvent e, JLayer<? extends V> l)**
- **void processFocusEvent(FocusEvent e, JLayer<? extends V> l)**
- **void processHierarchyBoundsEvent(HierarchyEvent e, JLayer<? extends V> l)**
- **void processHierarchyEvent(HierarchyEvent e, JLayer<? extends V> l)**

```
javax.swing.plaf.LayerUI<V extends Component> 7 (окончание)
```

- void processInputMethodEvent(InputMethodEvent e, JLayer<? extends V> l)
- void processKeyEvent(KeyEvent e, JLayer<? extends V> l)
- void processMouseEvent(MouseEvent e, JLayer<? extends V> l)
- void processMouseMotionEvent(MouseMotionEvent e, JLayer<? extends V> l)
- void processMouseWheelEvent(MouseWheelEvent e, JLayer<? extends V> l)

Вызываются при отправке указанного события данному объекту типа `LayerUI`.

В этой главе вы ознакомились с тем, как пользоваться усовершенствованными компонентами, предоставляемыми в библиотеке Swing. А в следующей главе мы обсудим вопросы, связанные с применением расширенных средств из библиотеки AWT, включая сложные операции рисования, манипулирование изображениями, печать и сопряжение с оконной системой конкретной платформы.

Расширенные средства AWT

В этой главе...

- ▶ Конвейер визуализации
- ▶ Фигуры
- ▶ Участки
- ▶ Обводка
- ▶ Раскраска
- ▶ Преобразование координат
- ▶ Отсечение
- ▶ Прозрачность и композиция
- ▶ Указания по воспроизведению
- ▶ Чтение и запись изображений
- ▶ Манипулирование изображениями
- ▶ Вывод изображений на печать
- ▶ Буфер обмена
- ▶ Перетаскивание объектов
- ▶ Интегрирование с платформой

Для создания простых рисунков можно использовать методы из класса `Graphics`. Они эффективны для разработки простых аплетов и приложений, но их недостаточно для построения сложных фигур или полного контроля над внешним видом рисунков. В состав прикладного программного интерфейса

Java 2D API входит библиотека классов для создания высококачественных рисунков. В этой главе рассматриваются расширенные средства этой библиотеки, а также обсуждаются вопросы вывода получаемых изображений на печать и способы реализации функций печати в прикладных программах.

Кроме того, рассматриваются два способа обмена данными между программами: с помощью буфера обмена и механизма перетаскивания. Эти способы можно использовать для передачи данных между двумя приложениями на Java или прикладной программой на Java и собственной программой для конкретной платформы. И наконец, в главе представлены методики разработки прикладных программ на Java в стиле собственных приложений для конкретной платформы, включая предоставление начального экрана при запуске программы и пиктограммы на панели задач операционной системы.

11.1. Конвейер визуализации

В исходной версии JDK 1.0 применялся очень простой механизм для рисования фигур. Для этого достаточно было выбрать нужный цвет, режим рисования и вызвать подходящие методы из класса `Graphics`, например `drawRect()` или `fillOval()`. В прикладном программном интерфейсе Java 2D API поддерживается намного больше возможностей для рисования двухмерной графики. Он, в частности, позволяет делать следующее.

- Легко формировать разные фигуры.
- Управлять *обводкой*, т.е. вычерчивать пером контуры фигур.
- Заполнять фигуры любым сплошным цветом, используя различные оттенки и узоры.
- Выполнять *преобразования* для перемещения, масштабирования, вращения и растягивания фигур.
- Отсекать фигуры таким образом, чтобы ограничить их произвольно выби-раемым участком экрана.
- Выбирать *правила композиции*, чтобы описывать порядок сочетания пиксельей новой и уже существующей фигур.
- Давать *указания по воспроизведению* для достижения компромисса между скоростью загрузки и качеством рисования.

Чтобы нарисовать фигуру, необходимо выполнить следующие действия.

1. Получить объект класса `Graphics2D`. Это подкласс, производный от класса `Graphics`. Начиная с версии Java SE 1.2 такие методы, как `paint()` и `paintComponent()`, автоматически получают объект класса `Graphics2D`. Поэтому остается только выполнить приведение типов, как показано ниже.

```
public void paintComponent(Graphics g)
{
    Graphics2D g2 = (Graphics2D) g;
    . . .
}
```

- Вызвать приведенный ниже метод `setRenderingHints()`, чтобы добавить указания по воспроизведению для достижения компромисса между скоростью и качеством рисования.

```
Rendering Hintshints = . . .;
g 2.setRendering Hints(hints);
```

- Вызвать приведенный ниже метод `setStroke()`, чтобы задать обводку контура фигуры. Для обводки можно выбрать толщину, а также сплошную или пунктирную линию.

```
Stroke stroke = . . .;
g 2.setStroke(stroke);
```

- Вызвать приведенный ниже метод `setPaint()`, чтобы указать способ раскраски. Раскраска подразумевает закрашивание участков контура обводки или внутренней области фигуры. Она может состоять из одного сплошного цвета, нескольких меняющихся оттенков или мозаичных узоров.

```
Paint paint = . . .;
g 2.setPaint(paint);
```

- Вызвать метод `clip()`, чтобы задать область отсечения следующим образом:

```
Shape clip = . . .;
g 2.clip(clip);
```

- Вызвать приведенный ниже метод `setTransform()` для преобразования рисунка из пользовательского пространства в пространство конкретного устройства. Такое преобразование следует выполнять в тех случаях, когда фигуру проще создать в специальной системе координат, чем использовать для этой цели координаты, выражаемые в пикселях.

```
AffineTransform transform = . . .;
g 2.transform(transform);
```

- Вызвать приведенный ниже метод `setComposite()`, чтобы задать правило композиции, описывающее порядок объединения новых пикселей с уже существующими пикселями.

```
Composite composite = . . .;
g 2.setComposite(composite);
```

- Создать фигуру, как показано ниже. В прикладном программном интерфейсе Java 2D API предусмотрено немало объектов фигур и методов для их сочетания.

```
Shape shape = . . .;
```

- Нарисовать или заполнить фигуру, как показано ниже. Под рисованием подразумевается очерчивание контуров фигуры, а под заполнением — закрашивание ее внутренней области.

```
g 2.draw(shape);
g 2.fill(shape);
```

Разумеется, на практике выполнять все эти действия обычно не требуется, и поэтому для многих параметров двухмерного графического контекста предусмотрены значения по умолчанию, изменять которые следует только в случае особой необходимости. В последующих разделах рассматриваются способы описания фигур, обводок, раскрасок, преобразований и правил композиции.

Различные методы типа `set()` просто устанавливают состояние двухмерного графического контекста и не относятся к конкретным рисункам. Аналогично фигура воспроизводится не во время создания объекта типа `Shape`, а при вызове метода `draw()` или `fill()`, когда новая фигура рассчитывается в конвейере визуализации (рис. 11.1).



Рис. 11.1. Конвейер визуализации

Для воспроизведения фигуры в конвейере визуализации выполняются следующие действия.

1. Обводится контур фигуры.
2. Выполняется преобразование фигуры.
3. Происходит отсечение фигуры. Данный процесс прекращается, как только фигура перестает пересекать область отсечения.
4. Заполняется часть фигуры, оставшаяся после отсечения.
5. Осуществляется композиция, т.е. составление пикселей фигуры и уже существующих пикселей в единый рисунок. (Как показано на рис. 11.1, существующие пиксели образуют круг, а фигура чашки накладывается на него.)

В следующем разделе сначала рассматриваются способы определения фигур, а затем порядок установки двухмерного графического контекста.

java.awt.Graphics2D 1.2

- **void draw(Shape s)**
Рисует контур указанной фигуры в соответствии с текущей раскраской.
- **void fill(Shape s)**
Заполняет внутреннюю область фигуры в соответствии с текущей раскраской.

11.2. Фигуры

Ниже перечислен ряд методов из класса `Graphics`, применяемых для рисования фигур.

`DrawLine()`
`drawRectangle()`

```
drawRoundRect()
draw3Drect()
drawPolygon()
drawPolyline()
drawOval()
drawArc()
```

Для них существуют соответствующие методы заполнения фигур, которые были предусмотрены в классе `Graphics`, начиная с версии JDK 1.0. В прикладном программном интерфейсе Java 2D API используется совершенно другая, объектно-ориентированная модель, где вместо методов применяются перечисленные ниже классы. Эти классы реализуют интерфейс `Shape` и рассматриваются в последующих разделах.

```
Line2D
Rectangle2D
RoundRectangle2D
Ellipse2D
Arc2D
QuadCurve2D
CubicCurve2D
GeneralPath
```

11.2.1. Иерархия классов рисования фигур

Классы `Line2D`, `Rectangle2D`, `RoundRectangle2D`, `Ellipse2D` и `Arc2D` соответствуют методам `drawLine()`, `drawRectangle()`, `drawRoundRect()`, `drawOval()` и `drawArc()`. Для понятия “трехмерный прямоугольник” не предусмотрено соответствующего метода `draw3DRect()`. Но в прикладном программном интерфейсе Java 2D API поддерживаются два дополнительных класса для рисования кривых второго и третьего порядков, рассматриваемых далее. Для рисования многоугольников также не предусмотрен отдельный класс вроде `Polygon2D`, а предлагается класс `GeneralPath`, описывающий контуры, состоящие из линий и кривых второго и третьего порядков. Класс `GeneralPath` можно использовать для построения многоугольников, как поясняется далее.

Чтобы нарисовать фигуру, необходимо сначала создать объект класса, реализующего интерфейс `Shape`, а затем вызвать метод `draw()` из класса `Graphics2D`. Перечисленные ниже классы наследуют общий класс `RectangularShape`. Как известно, эллипсы и дуги не являются прямоугольниками, но их можно вписать в ограничивающий прямоугольник (рис. 11.2).

```
Rectangle2D
RoundRectangle2D
Ellipse2D
Arc2D
```

У каждого из классов, название которого оканчивается на `2D`, имеются два подкласса для задания координат в виде числовых значений типа `float` и `double`. Так, в первом томе настоящего издания уже встречались обозначения `Rectangle2D.Float` и `Rectangle2D.Double`. Такая же схема используется для обозначения других классов, например `Arc2D.Float` и `Arc2D.Double`.

Для внутреннего представления координат во всех классах, предназначенных для построения графики, используются числовые данные типа `float`, поскольку

они требуют меньше места для хранения, чем числовые данные типа double. К тому же они поддерживают достаточную точность для геометрических расчетов. Но в языке Java обработка числовых данных типа float выполняется очень громоздкими и неуклюжими средствами. Поэтому большинство методов из классов для построения графики принимают параметры типа double и возвращают значения типа double. Выбирать тип числовых данных (float или double) и соответствующий конструктор класса приходится только при создании двухмерного объекта, как показано в следующем примере кода:

```
Rectangle2D floatRect = new Rectangle2D.Float(5F, 10F, 7.5F, 15F);
Rectangle2D doubleRect = new Rectangle2D.Double(5, 10, 7.5, 15);
```

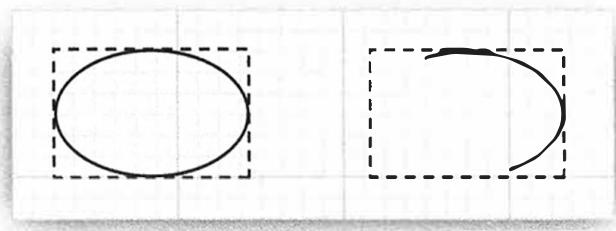


Рис. 11.2. Эллипс и дугу можно вписать
в ограничивающий прямоугольник

Классы Xxx2D.Float и Xxx2D.Double являются производными от классов Xxx2D. После создания объекта нет никакой нужды запоминать подклассы — можно просто сохранить созданный объект в переменной суперкласса, как показано в приведенном выше примере кода.

Судя по названиям классов Xxx2D.Float и Xxx2D.Double, можно сделать вывод, что они являются внутренними для классов Xxx2D. Это простое синтаксическое правило позволяет избежать чрезмерного увеличения длины имен внешних классов. И наконец, предусмотрен класс Point2D, описывающий точку с координатами *x* и *y*. Точки полезны для определения фигур, но сами они не являются фигурами.

На рис. 11.3 схематически представлены отношения наследования между классами рисования фигур без указания подклассов Double и Float. Устаревшие классы, унаследованные из прежних версий JDK, выделены серым цветом.

11.2.2. Применение классов рисования фигур

О классах Line2D, Rectangle2D и Ellipse2D уже упоминалось в главе 10 первого тома настоящего издания, а в этом разделе речь пойдет о том, как пользоваться классами для построения остальных двухмерных фигур. В частности, для построения прямоугольника со скругленными углами в конструкторе класса RoundRectangle2D следует указать координаты верхнего левого угла, ширину, высоту и размеры области скругления углов по осям *x* и *y* (рис. 11.4). Например, в результате следующего вызова конструктора данного класса создается прямоугольник с радиусом скругления углов, равным 20:

```
RoundRectangle2D r =
    new RoundRectangle2D.Double(150, 200, 100, 50, 20, 20);
```

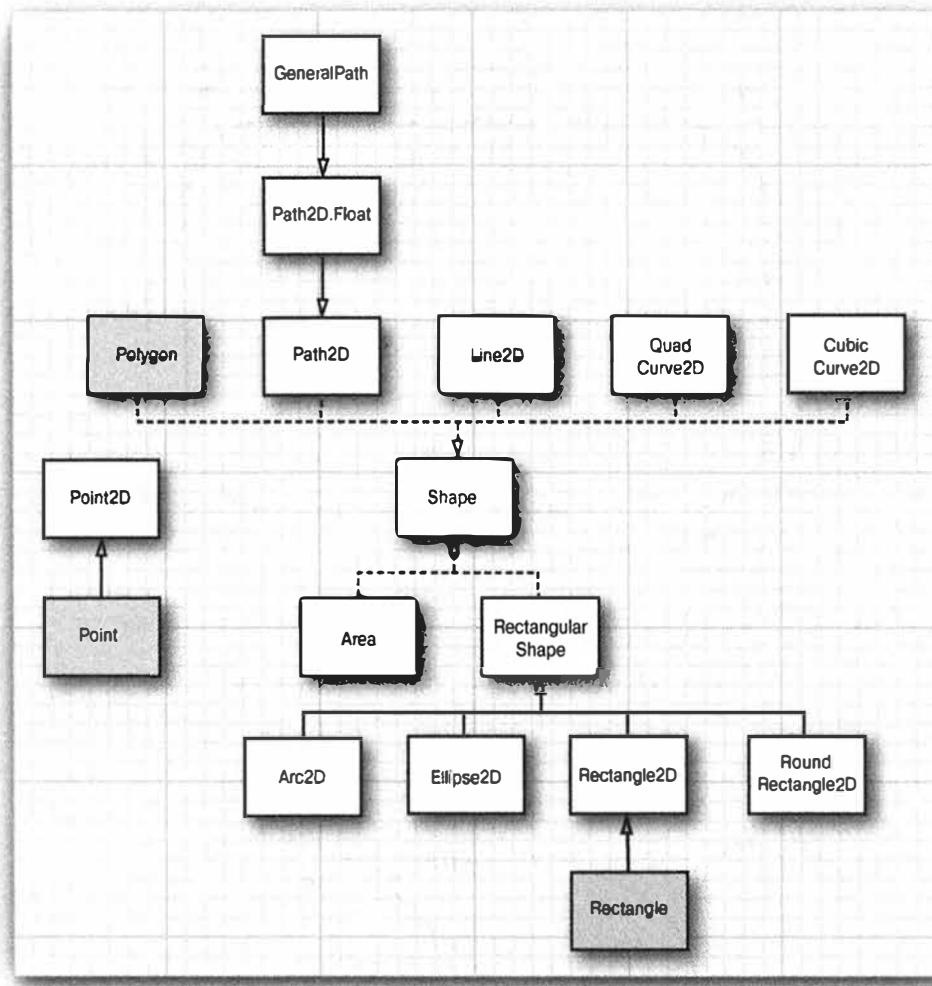


Рис. 11.3. Отношения наследования между классами рисования фигур

Для построения дуги следует указать ограничивающий прямоугольник, начальный угол, угол разворота дуги (рис. 11.5), а также один из видов замыкания дуги: `Arc2D.OPEN`, `Arc2D.PIE` или `Arc2D.CHORD`. Ниже приведен пример построения дуги, а на рис. 11.6 — виды замыкания дуги.

```
Arc2D a = new Arc2D(x, y, width, height, startAngle, arcAngle,
closureType);
```

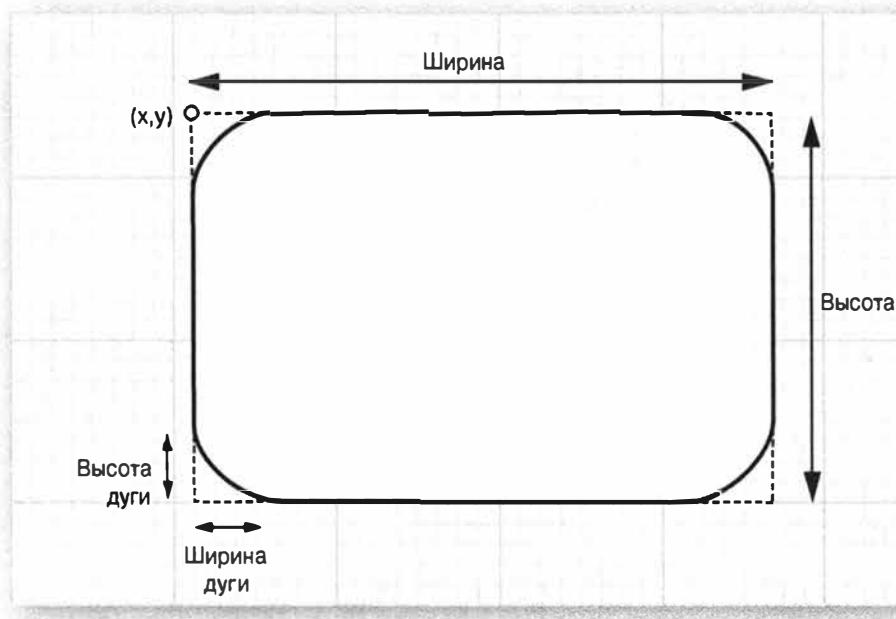


Рис. 11.4. Построение прямоугольника со скругленными углами средствами класса RoundRectangle2D

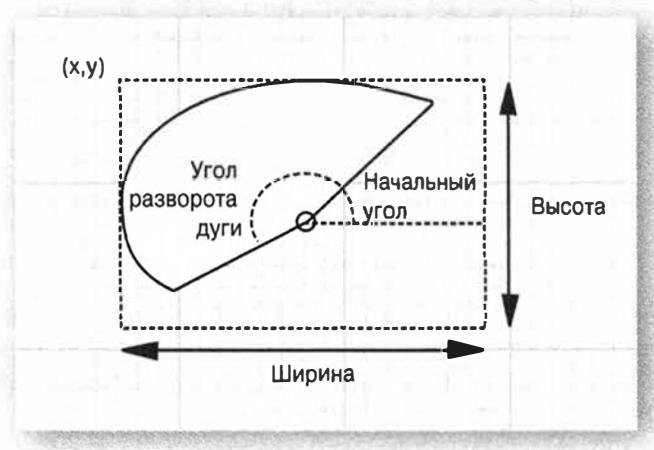


Рис. 11.5. Построение эллиптической дуги

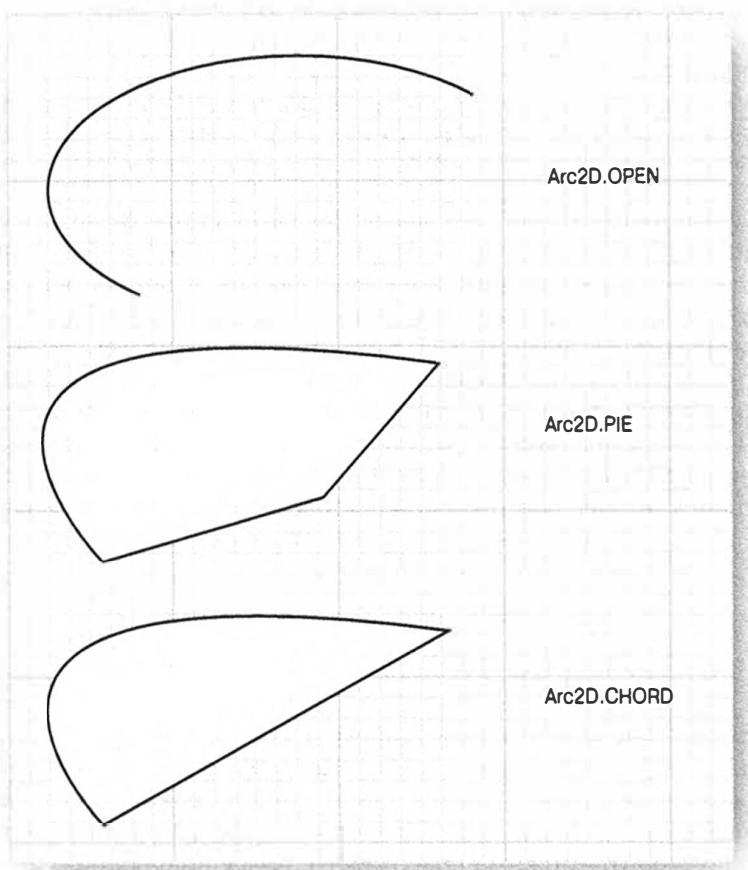


Рис. 11.6. Виды замыкания дуги



Внимание! Если дуга имеет эллиптическую форму, рассчитать ее углы не так-то просто. По этому поводу в документации на прикладной программный интерфейс API заявляется следующее: "Углы задаются относительно ограничивающего прямоугольника таким образом, чтобы линия, проведенная из центра эллипса к правому верхнему углу ограничивающего прямоугольника, образовывала угол 45° с обеими его сторонами. Если же обрамляющий прямоугольник заметно длиннее по одной оси, чем по другой, углы к началу и к концу отрезка дуги будут скашиваться вдоль более длинной оси ограничивающей рамки". К сожалению, в документации ничего не говорится о том, как рассчитывать такое скашивание. Поэтому покажем, как это делается на практике.

Допустим, центр дуги находится в точке начала отсчета, а точка с координатами $\{x, y\}$ — на самой дуге. Угол скашивания в этом случае рассчитывается следующим образом:

```
skewedAngle = Math.toDegrees(Math.atan2(-y * height, x * width));
```

В итоге получается значение в пределах от -180 до 180 . Подобным образом рассчитываются сначала начальный и конечный углы скашивания, а затем их разность. Если получится

отрицательное значение, то прибавляется значение **360**. Далее остается лишь предоставить конструктору дуги значения начального угла и разности двух углов срезывания в качестве параметров.

Запустив на выполнение пример программы, исходный код которой приведен в конце этого раздела, можете сами убедиться, что описанный выше порядок расчета углов срезывания действительно дает правильные значения параметров для конструктора дуги, как показано на рис. 11.9.

В прикладном программном интерфейсе Java 2D API предусмотрены средства для построения кривых *второго* порядка (квадратичных) и *третьего* порядка (кубических). Здесь не рассматриваются математические аспекты построения этих кривых, но для демонстрации их возможностей предлагается запустить на выполнение программу из листинга 11.1. Как показано на рис. 11.7 и 11.8, кривые второго и третьего порядка определяются двумя *конечными точками* и одной или двумя *управляющими точками* (для кривых второго и третьего порядка соответственно). При перемещении управляющих точек изменяется форма кривых. Для построения кривых второго и третьего порядка задаются координаты конечных и управляющих точек, как показано в приведенном ниже примере кода.

```
QuadCurve2D q = new QuadCurve2D.Double(startX, startY,
                                         controlX, controlY,
                                         endX, endY);
CubicCurve2D c = new CubicCurve2D.Double(startX, startY,
                                            control1X, control1Y,
                                            control2X, control2Y,
                                            endX, endY);
```

Кривые второго порядка не очень удобны, и поэтому они редко используются на практике, в отличие от кривых третьего порядка (например, кривых Безье, вычерчиваемых средствами класса *CubicCurve2D*). Сочетая несколько кривых третьего порядка таким образом, чтобы в точках соединения совпадали углы их наклона, можно строить сложные поверхности. Более подробно об этом можно узнать из книги *Computer Graphics: Principles and Practice, Second Edititon in C* Джеймса Фоли, Андриса ван Дама, Стивена Файнера и др. (James D. Foley, Andries van Dam, Steven K. Feiner et al.; издательство Addison Wesley, 1995 г.).

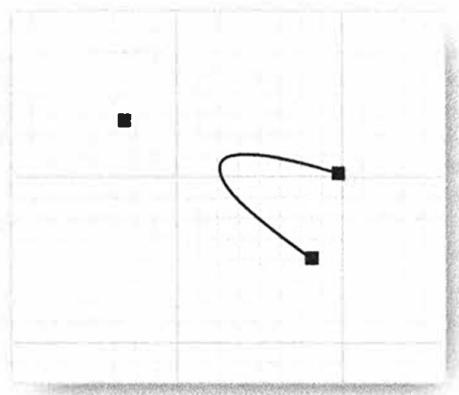


Рис. 11.7. Кривая второго порядка

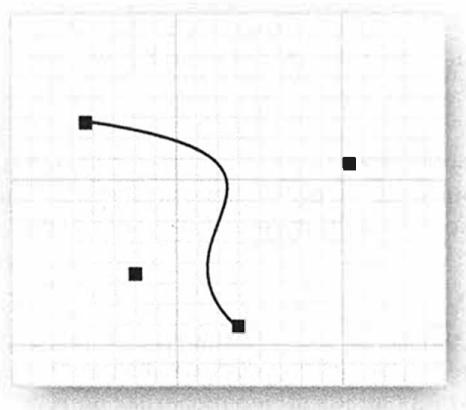


Рис. 11.8. Кривая третьего порядка

Для построения и сохранения произвольной кривой в виде последовательности линейных отрезков, кривых второго и третьего порядка предусмотрен класс `GeneralPath`. С этой целью методу `moveTo()` следует передать координаты первой точки контура строящейся кривой:

```
GeneralPath path = new GeneralPath();
path.moveTo(10, 20);
```

Далее контур строящейся кривой расширяется с помощью метода `lineTo()`, `quadTo()` или `curveTo()`. Эти методы продолжают контур строящейся кривой линией, кривой второго или третьего порядка соответственно. При вызове метода `lineTo()` следует указать конечную точку, а при вызове двух других методов построения кривых — сначала управляющие точки и затем конечную точку, как показано ниже. И наконец, контур строящейся кривой замыкается с помощью метода `closePath()`, где проводится линия в обратном направлении к начальной точке данного контура.

```
path.lineTo(20, 30);
path.curveTo(control1X, control1Y, control2X, control2Y,
            endX, endY);
```

Чтобы построить многоугольник, следует вызвать метод `moveTo()` для перехода к его первой вершине, а затем несколько раз вызвать метод `lineTo()` для соединения отрезками линии остальных вершин многоугольника. И наконец, остается вызвать метод `closePath()`, чтобы замкнуть многоугольник. Более подробно этот процесс демонстрируется далее в примере программы из листинга 11.1. Общий контур вычерчиваемой фигуры совсем не обязательно должен быть соединенным. Это означает, что метод `moveTo()` можно вызывать в любой удобный момент, чтобы начать построение нового отрезка контура.

И наконец, к общему контуру можно добавить произвольные объекты типа `Shape`, используя для этой цели метод `append()`. При этом контур присоединяемой фигуры добавляется в конце общего контура вычерчиваемой фигуры. Второй параметр этого метода принимает логическое значение `true`, если новую фигуру необходимо соединить с последней точкой общего контура,

а иначе — логическое значение `false`. Например, в приведенном ниже фрагменте кода контур прямоугольника добавляется к общему контуру вычерчиваемой фигуры, не соединяясь с этим контуром в его конце.

```
Rectangle2D r = . . .;
path.append(r, false);
```

Но при следующем вызове метода `append()` сначала проводится прямая линия, соединяющая конечную точку общего контура с начальной точкой прямоугольника `r`, а затем контур прямоугольника добавляется к общему контуру вычерчиваемой фигуры:

```
path.append(r, true);
```

На рис. 11.7 и 11.8 показаны результаты выполнения примера программы из листинга 11.1. В рабочем окне этой программы находится раскрывающийся список для выбора следующих видов вычерчиваемых фигур.

- Прямые линии.
- Прямоугольники, в том числе скругленные, а также эллипсы.
- Дуги (помимо самой дуги, отображаются контуры ограничивающего прямоугольника, маркер начального угла и угол разворота дуги).
- Многоугольники (вычерчиваемые средствами класса `GeneralPath`).
- Кривые второго и третьего порядка.

Изменить расположение управляющих точек можно с помощью мыши. Обратите внимание на то, что при перемещении этих точек фигура автоматически перерисовывается. Рассматриваемая здесь программа имеет довольно сложную структуру, поскольку она оперирует многими фигурами и поддерживает их автоматическую перерисовку при перетаскивании управляющих точек.

Абстрактный суперкласс `ShapeMaker` инкапсулирует общие функции всех классов для построения фигур. У каждой фигуры имеются управляющие точки, которые пользователь может перемещать, а метод `getPointCount()` возвращает их количество. Для расчета конкретной формы фигуры, исходя из текущего положения управляющих точек, служит следующий абстрактный метод:

```
Shape makeShape(Point2D[] points)
```

Метод `toString()` возвращает имя класса, и благодаря этому объекты типа `ShapeMaker` могут быть просто внесены в комбинированный список типа `JComboBox`. Для активизации режима перетаскивания управляющих точек в классе `ShapePanel` следует предусмотреть обработку событий от мыши. Так, если кнопка мыши нажата над прямоугольным маркером управляющей точки, этот маркер перемещается при последующем движении мыши.

Большинство классов для построения фигур имеют очень простую структуру. Их методы `makeShape()` строят и возвращают требующуюся фигуру. Но для применения класса `ArcMaker` приходится дополнительно рассчитывать начальный и конечный углы сглаживания, как пояснялось выше. Поэтому для демонстрации правильности выполненных расчетов возвращаемая фигура вычерчивается средствами класса `GeneralPath`, включая саму дугу, ограничивающий ее прямоугольник и линии, проведенные из центра дуги к управляющим точкам (рис. 11.9).

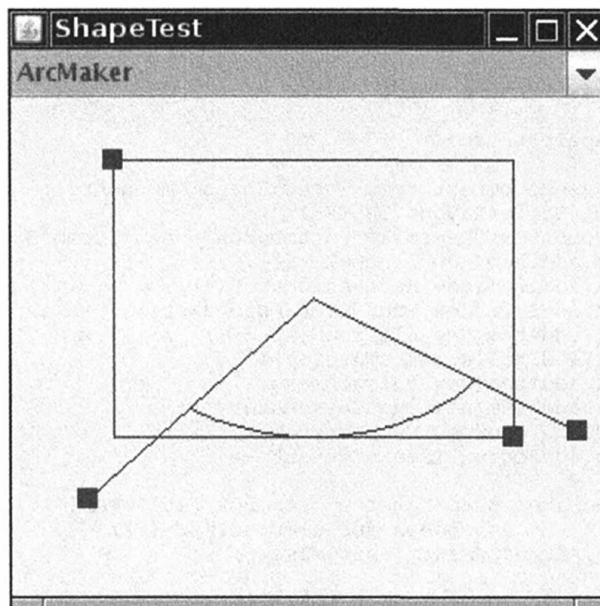


Рис. 11.9. Рабочее окно программы ShapeTest с построенной дугой и ее вспомогательными элементами

Листинг 11.1. Исходный код из файла shape/ShapeTest.java

```

1 package shape;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.awt.geom.*;
6 import java.util.*;
7 import javax.swing.*;
8
9 /**
10  * В этой программе демонстрируется построение
11  * различных двухмерных фигур
12  * @version 1.03 2016-05-10
13  * @author Cay Horstmann
14  */
15 public class ShapeTest
16 {
17     public static void main(String[] args)
18     {
19         EventQueue.invokeLater(() ->
20         {
21             JFrame frame = new ShapeTestFrame();
22             frame.setTitle("ShapeTest");
23             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24             frame.setVisible(true);
25         });
26     }
27 }
28 /**
29 */

```

```
30 * Этот фрейм содержит комбинированный список для выбора фигур,
31 * а также компонент для их рисования
32 */
33 class ShapeTestFrame extends JFrame
34 {
35     public ShapeTestFrame()
36     {
37         final ShapeComponent comp = new ShapeComponent();
38         add(comp, BorderLayout.CENTER);
39         final JComboBox<ShapeMaker> comboBox = new JComboBox<>();
40         comboBox.addItem(new LineMaker());
41         comboBox.addItem(new RectangleMaker());
42         comboBox.addItem(new RoundRectangleMaker());
43         comboBox.addItem(new EllipseMaker());
44         comboBox.addItem(new ArcMaker());
45         comboBox.addItem(new PolygonMaker());
46         comboBox.addItem(new QuadCurveMaker());
47         comboBox.addItem(new CubicCurveMaker());
48         comboBox.addActionListener(event ->
49             {
50                 ShapeMaker shapeMaker = comboBox.getItemAt(
51                     comboBox.getSelectedIndex());
52                 comp.setShapeMaker(shapeMaker);
53             });
54         add(comboBox, BorderLayout.NORTH);
55         comp.setShapeMaker((ShapeMaker) comboBox.getItemAt(0));
56         pack();
57     }
58 }
59 /**
60 * Этот компонент рисует фигуру и дает пользователю
61 * возможность перемещать определяющие ее точки
62 */
63 class ShapeComponent extends JComponent
64 {
65     private static final Dimension PREFERRED_SIZE =
66         new Dimension(300, 200);
67     private Point2D[] points;
68     private static Random generator = new Random();
69     private static int SIZE = 10;
70     private int current;
71     private ShapeMaker shapeMaker;
72
73     public ShapeComponent()
74     {
75         addMouseListener(new MouseAdapter()
76             {
77                 public void mousePressed(MouseEvent event)
78                 {
79                     Point p = event.getPoint();
80                     for (int i = 0; i < points.length; i++)
81                     {
82                         double x = points[i].getX() - SIZE / 2;
83                         double y = points[i].getY() - SIZE / 2;
84                         Rectangle2D r =
85                             new Rectangle2D.Double(x, y, SIZE, SIZE);
86                         if (r.contains(p))
87                         {
88                             current = i;
89                             return;
```

```
91         }
92     }
93 }
94
95     public void mouseReleased(MouseEvent event)
96     {
97         current = -1;
98     }
99 });
100 addMouseMotionListener(new MouseMotionAdapter()
101 {
102     public void mouseDragged(MouseEvent event)
103     {
104         if (current == -1) return;
105         points[current] = event.getPoint();
106         repaint();
107     }
108 });
109 current = -1;
110 }
111
112 /**
113 * Устанавливает построитель фигур и инициализирует
114 * его произвольным рядом точек
115 * @param aShapeMaker Построитель фигур, определяющий фигуру
116 * по ряду исходных точек
117 */
118 public void setShapeMaker(ShapeMaker aShapeMaker)
119 {
120     shapeMaker = aShapeMaker;
121     int n = shapeMaker.getPointCount();
122     points = new Point2D[n];
123     for (int i = 0; i < n; i++)
124     {
125         double x = generator.nextDouble() * getWidth();
126         double y = generator.nextDouble() * getHeight();
127         points[i] = new Point2D.Double(x, y);
128     }
129     repaint();
130 }
131
132 public void paintComponent(Graphics g)
133 {
134     if (points == null) return;
135     Graphics2D g2 = (Graphics2D) g;
136     for (int i = 0; i < points.length; i++)
137     {
138         double x = points[i].getX() - SIZE / 2;
139         double y = points[i].getY() - SIZE / 2;
140         g2.fill(new Rectangle2D.Double(x, y, SIZE, SIZE));
141     }
142
143     g2.draw(shapeMaker.makeShape(points));
144 }
145
146 public Dimension getPreferredSize() { return PREFERRED_SIZE; }
147 }
148
149 /**
150 * Построитель фигур по ряду исходных точек. Конкретные
151 * подклассы должны возвращать форму из метода makeShape()
```

```
152 */
153 abstract class ShapeMaker
154 {
155     private int pointCount;
156
157 /**
158 * Конструирует построитель фигур
159 * @param pointCount Количество исходных точек, требующихся
160 *                   для определения данной фигуры
161 */
162 public ShapeMaker(int pointCount)
163 {
164     this.pointCount = pointCount;
165 }
166
167 /**
168 * Получает количество исходных точек, требующихся
169 * для определения данной фигуры
170 * @return Возвращает количество исходных точек
171 */
172 public int getPointCount()
173 {
174     return pointCount;
175 }
176
177 /**
178 * Строит фигуру по заданному ряду исходных точек
179 * @param p Исходные точки, определяющие фигуру
180 * @return Возвращает Фигуру, определяемую исходными точками
181 */
182 public abstract Shape makeShape(Point2D[] p);
183
184 public String toString()
185 {
186     return getClass().getName();
187 }
188 }
189
190 /**
191 * Проводит линию, соединяющую две заданные точки
192 */
193 class LineMaker extends ShapeMaker
194 {
195     public LineMaker()
196     {
197         super(2);
198     }
199
200     public Shape makeShape(Point2D[] p)
201     {
202         return new Line2D.Double(p[0], p[1]);
203     }
204 }
205
206 /**
207 * Строит прямоугольник, соединяющий две заданные угловые точки
208 */
209 class RectangleMaker extends ShapeMaker
210 {
211     public RectangleMaker()
212     {
```

```
213     super(2);
214 }
215
216 public Shape makeShape(Point2D[] p)
217 {
218     Rectangle2D s = new Rectangle2D.Double();
219     s.setFrameFromDiagonal(p[0], p[1]);
220     return s;
221 }
222 }
223
224 /**
225 * Строит прямоугольник со скругленными углами,
226 * соединяющий две заданные угловые точки
227 */
228 class RoundRectangleMaker extends ShapeMaker
229 {
230     public RoundRectangleMaker()
231     {
232         super(2);
233     }
234
235     public Shape makeShape(Point2D[] p)
236     {
237         RoundRectangle2D s = new RoundRectangle2D.Double(
238             0, 0, 0, 0, 20, 20);
239         s.setFrameFromDiagonal(p[0], p[1]);
240         return s;
241     }
242 }
243
244 /**
245 * Строит эллипс, вписанный в ограничивающий прямоугольник
246 * с двумя заданными угловыми точками
247 */
248 class EllipseMaker extends ShapeMaker
249 {
250     public EllipseMaker()
251     {
252         super(2);
253     }
254
255     public Shape makeShape(Point2D[] p)
256     {
257         Ellipse2D s = new Ellipse2D.Double();
258         s.setFrameFromDiagonal(p[0], p[1]);
259         return s;
260     }
261 }
262
263 /**
264 * Строит дугу, вписанную в ограничивающий прямоугольник
265 * с двумя заданными угловыми точками. Начальный и конечный
266 * углы дуги задают линии, проведенные из центра ограничивающего
267 * прямоугольника в две заданные точки. Для демонстрации
268 * правильности расчета углов возвращаемая фигура состоит
269 * из дуги, ограничивающего прямоугольника и линий,
270 * образующих эти углы
271 */
272 class ArcMaker extends ShapeMaker
273 {
```

```
274 public ArcMaker()
275 {
276     super(4);
277 }
278
279 public Shape makeShape(Point2D[] p)
280 {
281     double centerX = (p[0].getX() + p[1].getX()) / 2;
282     double centerY = (p[0].getY() + p[1].getY()) / 2;
283     double width = Math.abs(p[1].getX() - p[0].getX());
284     double height = Math.abs(p[1].getY() - p[0].getY());
285
286     double skewedStartAngle = Math.toDegrees(
287         Math.atan2(-(p[2].getY() - centerY) * width,
288                     (p[2].getX() - centerX) * height));
289     double skewedEndAngle = Math.toDegrees(
290         Math.atan2(-(p[3].getY() - centerY) * width,
291                     (p[3].getX() - centerX) * height));
292     double skewedAngleDifference =
293         skewedEndAngle - skewedStartAngle;
294     if (skewedStartAngle < 0) skewedStartAngle += 360;
295     if (skewedAngleDifference < 0) skewedAngleDifference += 360;
296
297     Arc2D s = new Arc2D.Double(0, 0, 0, 0, skewedStartAngle,
298                                 skewedAngleDifference, Arc2D.OPEN);
299     s.setFrameFromDiagonal(p[0], p[1]);
300
301     GeneralPath g = new GeneralPath();
302     g.append(s, false);
303     Rectangle2D r = new Rectangle2D.Double();
304     r.setFrameFromDiagonal(p[0], p[1]);
305     g.append(r, false);
306     Point2D center = new Point2D.Double(centerX, centerY);
307     g.append(new Line2D.Double(center, p[2]), false);
308     g.append(new Line2D.Double(center, p[3]), false);
309     return g;
310 }
311 }
312
313 /**
314  * Строит многоугольник, определяемый шестью угловыми точками
315  */
316 class PolygonMaker extends ShapeMaker
317 {
318     public PolygonMaker()
319     {
320         super(6);
321     }
322
323     public Shape makeShape(Point2D[] p)
324     {
325         GeneralPath s = new GeneralPath();
326         s.moveTo((float) p[0].getX(), (float) p[0].getY());
327         for (int i = 1; i < p.length; i++)
328             s.lineTo((float) p[i].getX(), (float) p[i].getY());
329         s.closePath();
330         return s;
331     }
332 }
333
334 /**
```

```

335 * Строит кривую второго порядка, определяемую двумя
336 * конечными точками и одной управляющей точкой
337 */
338 class QuadCurveMaker extends ShapeMaker
339 {
340     public QuadCurveMaker()
341     {
342         super(3);
343     }
344
345     public Shape makeShape(Point2D[] p)
346     {
347         return new QuadCurve2D.Double(p[0].getX(), p[0].getY(),
348                                         p[1].getX(), p[1].getY(),
349                                         p[2].getX(), p[2].getY());
350     }
351 }
352 /**
353 * Строит кривую третьего порядка, определяемую двумя
354 * конечными точками и двумя управляющими точками
355 */
356 */
357 class CubicCurveMaker extends ShapeMaker
358 {
359     public CubicCurveMaker()
360     {
361         super(4);
362     }
363
364     public Shape makeShape(Point2D[] p)
365     {
366         return new CubicCurve2D.Double(p[0].getX(), p[0].getY(),
367                                         p[1].getX(), p[1].getY(),
368                                         p[2].getX(), p[2].getY(),
369                                         p[3].getX(), p[3].getY());
370     }
371 }

```

java.awt.geom.RoundRectangle2D.Double 1.2

- **RoundRectangle2D.Double(double x, double y, double width, double height, double arcWidth, double arcHeight)**

Строит прямоугольник со скругленными углами, исходя из заданных размеров ограничивающего прямоугольника и дуги скругления. Наглядное представление о параметрах **arcWidth** и **arcHeight** дает рис. 11.4.

java.awt.geom.Arc2D.Double 1.2

- **Arc2D.Double(double x, double y, double w, double h, double startAngle, double arcAngle, int type)**

Строит дугу, исходя из заданного ограничивающего прямоугольника, начального и конечного угла, а также вида замыкания дуги. Наглядное представление о параметрах **startAngle** и **arcAngle** дает рис. 11.5. Что касается параметра **type**, то он может принимать значение одной из следующих констант: **Arc2D.OPEN**, **Arc2D.PIE** или **Arc2D.CHORD**.

java.awt.geom.QuadCurve2D.Double 1.2

- QuadCurve2D.Double(double x1, double y1, double ctrlx,
double ctrly, double x2, double y2)**

Строит кривую второго порядка по начальной, управляющей и конечной точкам.

java.awt.geom.CubicCurve2D.Double 1.2

- CubicCurve2D.Double(double x1, double y1, double ctrlx1,
double ctrly1, double ctrlx2, double ctrly2, double x2, double y2)**

Создает кривую третьего порядка по начальной, двум контрольным и конечной точкам.

java.awt.geom.GeneralPath 1.2

- GeneralPath()**

Строит пустой общий контур

java.awt.geom.Path2D.Float 6

- void moveTo(float x, float y)**

Строит текущую точку с координатами **x** и **y**, т.е. начальную точку следующего отрезка линии.

- void lineTo(float x, float y)**

- void quadTo(float ctrlx, float ctrly, float x, float y)**

- void curveTo(float ctrlx1, float ctrly1, float ctrlx2, float ctrly2,
float x, float y)**

Рисуют прямую линию, кривую второго или третьего порядка от текущей до конечной точки с координатами **x** и **y**, после чего конечная точка становится текущей.

java.awt.geom.Path2D 6

- void append(Shape s, boolean connect)**

Присоединяет контур заданной фигуры к общему контуру. Если параметр **connect** принимает логическое значение **true**, текущая точка общего контура соединяется прямой линией с начальной точкой присоединяемой фигуры.

- void closePath()**

Замыкает контур, рисуя прямую линию от текущей точки к первой точке контура.

11.3. Участки

В предыдущем разделе обсуждались средства построения сложных фигур с помощью прямых линий и кривых второго и третьего порядка. Используя

достаточное количество линий и кривых, можно нарисовать практически любую фигуру. Например, контуры символов в шрифтах, которые отображаются на экране и на отпечатке, состоят из прямых линий и кривых третьего порядка.

Но иногда описать фигуру легче, составляя ее из *участков*, например, прямоугольных, многоугольных или овальных. В прикладном программном интерфейсе Java 2D API поддерживаются четыре метода, выполняющие геометрические операции построения нового участка из двух исходных участков. Результаты выполнения этих операций приведены на рис. 11.10.

- Метод `add()` составляет участок, который содержит все точки первого или второго исходного участка. Это операция геометрического сложения участков.
- Метод `subtract()` составляет участок, который содержит все точки первого исходного участка, не входящие во второй исходный участок. Это операция геометрического вычитания участков.
- Метод `intersect()` составляет участок, который содержит только точки, принадлежащие одновременно первому и второму исходному участкам. Это операция геометрического пересечения участков.
- Метод `exclusiveOr()` составляет участок, который содержит все точки, принадлежащие первому или второму исходному участку, но не обоим вместе. Это операция геометрического исключения участков.

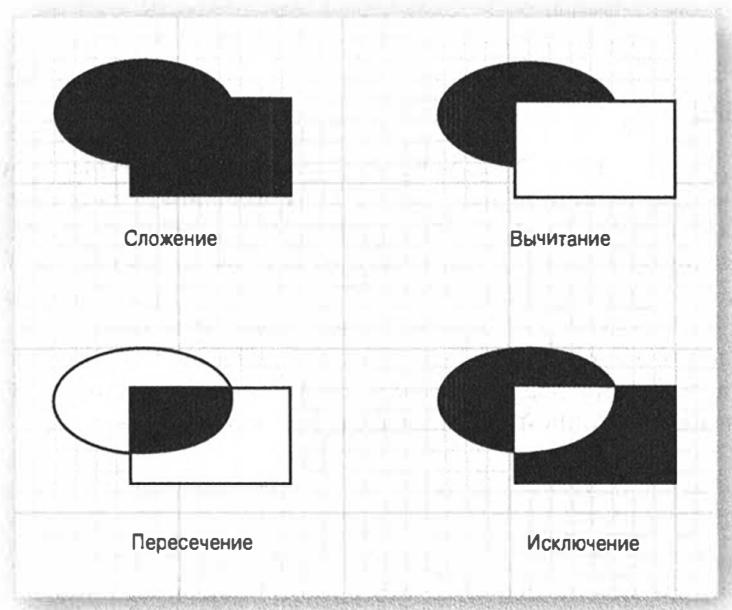


Рис. 11.10. Геометрические операции построения нового участка из двух исходных участков

Чтобы построить участок сложной геометрической формы, необходимо создать сначала исходный участок:

```
Area a = new Area();
```

Затем этот участок следует объединить с фигурой требующейся формы, выполнив одну из упомянутых выше геометрических операций:

```
a.add(new Rectangle2D.Double(. . .));
a.subtract(path);
. . .
```

Класс `Area` для построения участков реализует интерфейс `Shape`. В процессе построения участка можно очертить его границы с помощью метода `draw()` или заполнить его внутреннюю часть с помощью метода `fill()` из класса `Graphics2D`.

`java.awt.geom.Area`

- `void add(Area other)`
- `void subtract(Area other)`
- `void intersect(Area other)`
- `void exclusiveOr(Area other)`
- Выполняют геометрические операции построения нового участка, объединяя текущий участок с другим участком, определяемым параметром `other`. Получающийся в итоге участок становится текущим.

11.4. Обводка

Метод `draw()` из класса `Graphics2D` рисует границу фигуры, используя выбранный в настоящий момент вид обводки. По умолчанию для обводки выбирается сплошная линия толщиной в один пиксель. Для изменения вида обводки предусмотрен метод `setStroke()`, которому в качестве параметра передается экземпляр класса, реализующего интерфейс `Stroke`. В прикладном программном интерфейсе Java 2D API определен только один такой класс — `BasicStroke`. В этом разделе рассматриваются функциональные возможности данного класса.

Для обводки можно задать линию произвольной толщины. Например, для выбора линии толщиной 10 пикселей и последующей обводки служит следующий код:

```
g2.setStroke(new BasicStroke(10.0F));
g2.draw(new Line2D.Double(. . .));
```

Если толщина линии обводки превышает один пиксель, то для формирования концов линии можно использовать один из перечисленных ниже стилей (рис. 11.11).

- **Торцевой конец линии.** Линия обводки обрывается в конечной точке.
- **Скругленный конец линии.** В конце линии обводки добавляется полукруг.
- **Квадратный конец линии.** В конце линии обводки добавляется полуквадрат.

Для соединения двух толстых линий обводки можно также указать один из следующих стилей (рис. 11.12).

- **Косой стык.** Соединяет линии обводки по прямой линии, перпендикулярной биссектрисе угла между ними.
- **Скругленный стык.** Соединяет линии обводки, образуя скругленный конец линии.
- **Угловой стык.** Соединяет линии обводки, образуя клинообразный конец линии.



Рис. 11.11. Стили окончания линии обводки



Рис. 11.12. Стили соединения линий обводки

Угловой стык не совсем подходит для линий обводки, которые пересекаются под небольшими углами. Так, если две линии обводки пересекаются под углами, меньшими предела среза, то вместо углового используется косой стык. Благодаря этому предотвращается образование чрезмерно длинных клиньев. По умолчанию предел среза составляет 10° . Стили окончания и соединения линий обводки можно указать в конструкторе класса `BasicStroke`, как показано ниже.

```
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_ROUND,
                           BasicStroke.JOIN_ROUND));
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_BUTT,
                           BasicStroke.JOIN_MITER, 15.0F /* miter limit */));
```

И наконец, для пунктирных линий обводки можно выбрать конкретный *пунктир*. В примере программы из листинга 11.2 используется пунктир, соответствующий сигналу SOS в азбуке Морзе. Пунктир задается в виде массива типа `float[]`, который содержит длины рисуемых и пробельных отрезков обводки (рис. 11.13).

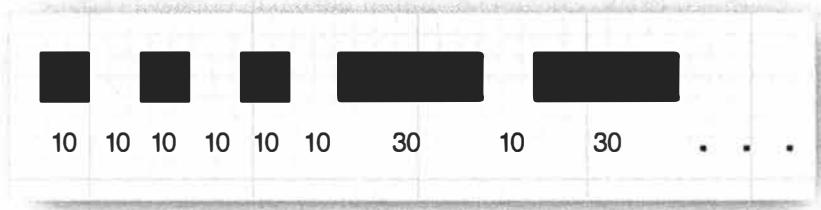


Рис. 11.13. Пунктир для обводки

При создании объекта типа `BasicStroke`, помимо самого пунктира, указывается также *фаза пунктира*, т.е. место начала пунктира на линии обводки. Обычно фаза пунктира устанавливается равной нулю. В приведенном ниже примере кода показано, каким образом задается пунктирный стиль линий обводки.

```
float[] dashPattern = { 10, 10, 10, 10, 10, 10, 30, 10, 30, ... };
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_BUTT,
    BasicStroke.JOIN_MITER, 10.0F /* предел среза */,
    dashPattern, 0 /* фаза пунктира */));
```

 **На заметку!** Стили окончания линий распространяются и на все штрихи пунктира.

В примере программы, исходный код которой приведен в листинге 11.2, предоставляется возможность указать стиль окончания линий обводки, стиль их соединения и пунктир (рис. 11.14). Кроме того, концы линий обводки можно перемещать, чтобы проверить действенность предела среза. Для этого следует выбрать угловой стык, а затем перетащить мышью отрезок линии обводки таким образом, чтобы две линии образовали очень острый угол. По достижениям предела среза угловой стык автоматически превращается в косой.

Рассматриваемая здесь программа похожа на программу из листинга 11.1. Обработчик событий от мыши реагирует на щелчок кнопкой мыши в конечной точке линии обводки, и при перемещении курсора он будет передвигать вслед за ним конец линии обводки. Группа кнопок-переключателей уведомляет о выборе пользователем стиля окончания и соединения линии обводки, а также сплошной или пунктирной линии. Метод `paintComponent()` из класса `StrokePanel` используется для построения общего конфигурации типа `GeneralPath` из двух отрезков, соединяющих три точки, которые пользователь может перемещать с помощью мыши. Затем создается объект типа `BasicStroke`, исходя из выбора, сделанного пользователем. И наконец, линия обводки рисуется по сформированному конфигуру.

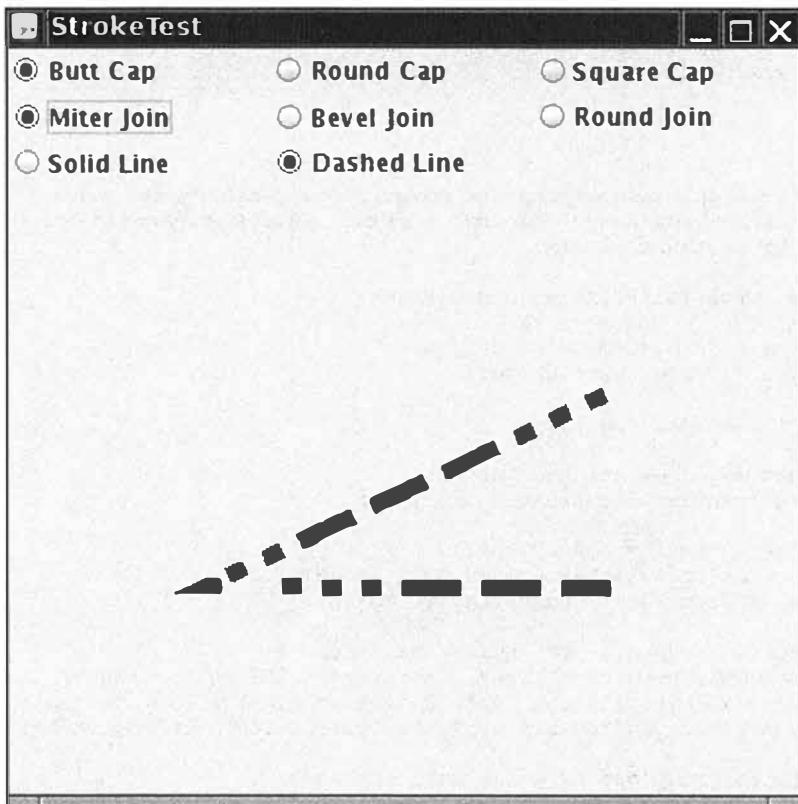


Рис. 11.14. Рабочее окно программы StrokeTest

Листинг 11.2. Исходный код из файла stroke/StrokeTest.java

```
1 package stroke;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.awt.geom.*;
6 import javax.swing.*;
7
8 /**
9  * В этой программе демонстрируются различные виды обводки
10 * @version 1.04 2016-05-10
11 * @author Cay Horstmann
12 */
13 public class StrokeTest
14 {
15     public static void main(String[] args)
16     {
17         EventQueue.invokeLater(() ->
18         {
19             JFrame frame = new StrokeTestFrame();
20             frame.setTitle("StrokeTest");
```

```
21         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22         frame.setVisible(true);
23     });
24 }
25 }
26
27 /**
28 * В этом фрейме пользователь может выбрать стиль окончания
29 * и соединения линий обводки, а также увидеть получающуюся
30 * в итоге линию обводки
31 */
32 class StrokeTestFrame extends JFrame
33 {
34     private StrokeComponent canvas;
35     private JPanel buttonPanel;
36
37     public StrokeTestFrame()
38     {
39         canvas = new StrokeComponent();
40         add(canvas, BorderLayout.CENTER);
41
42         buttonPanel = new JPanel();
43         buttonPanel.setLayout(new GridLayout(3, 3));
44         add(buttonPanel, BorderLayout.NORTH);
45
46         ButtonGroup group1 = new ButtonGroup();
47         makeCapButton("Butt Cap", BasicStroke.CAP_BUTT, group1);
48         makeCapButton("Round Cap", BasicStroke.CAP_ROUND, group1);
49         makeCapButton("Square Cap", BasicStroke.CAP_SQUARE, group1);
50
51         ButtonGroup group2 = new ButtonGroup();
52         makeJoinButton("Miter Join", BasicStroke.JOIN_MITER, group2);
53         makeJoinButton("Bevel Join", BasicStroke.JOIN_BEVEL, group2);
54         makeJoinButton("Round Join", BasicStroke.JOIN_ROUND, group2);
55
56         ButtonGroup group3 = new ButtonGroup();
57         makeDashButton("Solid Line", false, group3);
58         makeDashButton("Dashed Line", true, group3);
59     }
60
61 /**
62 * Создает кнопку-переключатель для выбора стиля
63 * окончания линии
64 * @param label Метка кнопки-переключателя
65 * @param style Стиль окончания линии
66 * @param group Группа кнопок-переключателей
67 */
68 private void makeCapButton(String label, final int style,
69                           ButtonGroup group)
70 {
71     // выбрать первую кнопку-переключатель в группе
72     boolean selected = group.getButtonCount() == 0;
73     JRadioButton button = new JRadioButton(label, selected);
74     buttonPanel.add(button);
75     group.add(button);
76     button.addActionListener(event -> canvas.setCap(style));
77     pack();
78 }
```

```
79
80 /**
81 * Создает кнопку-переключатель для выбора стиля
82 * соединения линий
83 * @param label Метка кнопки-переключателя
84 * @param style Стиль соединения линий
85 * @param group Группа кнопок-переключателей
86 */
87 private void makeJoinButton(String label, final int style,
88                             ButtonGroup group)
89 {
90     // выбрать первую кнопку-переключатель в группе
91     boolean selected = group.getButtonCount() == 0;
92     JRadioButton button = new JRadioButton(label, selected);
93     buttonPanel.add(button);
94     group.add(button);
95     button.addActionListener(event -> canvas.setJoin(style));
96 }
97
98 /**
99 * Создает кнопку-переключатель для выбора сплошных или
100 * пунктирных линий обводки
101 * @param label Метка кнопки-переключателя
102 * @param style Принимает логическое значение false,
103 *               если линия сплошная, или логическое
104 *               значение true, если линия пунктирная
105 */
106 private void makeDashButton(String label,
107                             final boolean style, ButtonGroup group)
108 {
109     // выбрать первую кнопку-переключатель в группе
110     boolean selected = group.getButtonCount() == 0;
111     JRadioButton button = new JRadioButton(label, selected);
112     buttonPanel.add(button);
113     group.add(button);
114     button.addActionListener(event -> canvas.setDash(style));
115 }
116 }
117
118 /**
119 * Этот компонент рисует две соединенные линии, используя
120 * разные объекты обводки и давая пользователю возможность
121 * перетаскивать три точки, определяющие линии обводки
122 */
123 class StrokeComponent extends JComponent
124 {
125     private static final Dimension PREFERRED_SIZE =
126             new Dimension(400, 400);
127     private static int SIZE = 10;
128
129     private Point2D[] points;
130     private int current;
131     private float width;
132     private int cap;
133     private int join;
134     private boolean dash;
135
136     public StrokeComponent()
```

```
137  {
138      addMouseListener(new MouseAdapter()
139      {
140          public void mousePressed(MouseEvent event)
141          {
142              Point p = event.getPoint();
143              for (int i = 0; i < points.length; i++)
144              {
145                  double x = points[i].getX() - SIZE / 2;
146                  double y = points[i].getY() - SIZE / 2;
147                  Rectangle2D r =
148                      new Rectangle2D.Double(x, y, SIZE, SIZE);
149                  if (r.contains(p))
150                  {
151                      current = i;
152                      return;
153                  }
154              }
155          }
156      }
157
158      public void mouseReleased(MouseEvent event)
159      {
160          current = -1;
161      }
162  });
163
164  addMouseMotionListener(new MouseMotionAdapter()
165  {
166      public void mouseDragged(MouseEvent event)
167      {
168          if (current == -1) return;
169          points[current] = event.getPoint();
170          repaint();
171      }
172  });
173
174  points = new Point2D[3];
175  points[0] = new Point2D.Double(200, 100);
176  points[1] = new Point2D.Double(100, 200);
177  points[2] = new Point2D.Double(200, 200);
178  current = -1;
179  width = 8.0F;
180 }
181
182  public void paintComponent(Graphics g)
183  {
184      Graphics2D g2 = (Graphics2D) g;
185      GeneralPath path = new GeneralPath();
186      path.moveTo((float) points[0].getX(),
187                  (float) points[0].getY());
188      for (int i = 1; i < points.length; i++)
189          path.lineTo((float) points[i].getX(),
190                      (float) points[i].getY());
191      BasicStroke stroke;
192      if (dash)
193      {
194          float miterLimit = 10.0F;
195          float[] dashPattern = { 10F, 10F, 10F, 10F, 10F,
```

```
196                     30F, 10F, 30F, 10F, 30F, 10F,
197                     10F, 10F, 10F, 10F, 30F };
198         float dashPhase = 0;
199         stroke = new BasicStroke(width, cap, join, miterLimit,
200                               dashPattern, dashPhase);
201     }
202     else stroke = new BasicStroke(width, cap, join);
203     g2.setStroke(stroke);
204     g2.draw(path);
205 }
206
207 /**
208 * Устанавливает стиль соединения линий
209 * @param j Стиль соединения линий
210 */
211 public void setJoin(int j)
212 {
213     join = j;
214     repaint();
215 }
216
217 /**
218 * Устанавливает стиль окончания линий
219 * @param c Стиль окончания линий
220 */
221 public void setCap(int c)
222 {
223     cap = c;
224     repaint();
225 }
226
227 /**
228 * Устанавливает сплошные или пунктирные линии
229 * @param d Принимает логическое значение false,
230 *          если линии сплошные, или логическое
231 *          значение true, если линии пунктирные
232 */
233 public void setDash(boolean d)
234 {
235     dash = d;
236     repaint();
237 }
238
239 public Dimension getPreferredSize() { return PREFERRED_SIZE; }
240 }
```

java.awt.Graphics2D 1.2**• void setStroke(Stroke s)**

Устанавливает в качестве обводки для данного графического контекста указанный объект, класс которого реализует интерфейс **Stroke**.

java.awt.BasicStroke 1.2

- **BasicStroke(float width)**
- **BasicStroke(float width, int cap, int join)**
- **BasicStroke(float width, int cap, int join, float miterlimit)**
- **BasicStroke(float width, int cap, int join, float miterlimit, float[] dash, float dashPhase)**

Конструируют объект обводки с заданными свойствами.

Параметры:

width	Толщина обводки
cap	Стиль окончания линии: CAP_BUTT , CAP_ROUND или CAP_SQUARE
join	Стиль соединения линий: JOIN_BEVEL , JOIN_MITER или JOIN_ROUND
miterlimit	Предел среза (в градусах), ниже которого угловой стык превращается в косой
dash	Пунктир в виде массива длин отображаемых и пробельных отрезков обводки
dashPhase	Фаза пунктира — длина отрезка, предшествующего первой точке линии обводки, при условии, что пунктир уже применяется до этой точки

11.5. Раскраска

Внутренняя часть фигуры заполняется путем *раскраски*. Для установки стиля раскраски служит объект, класс которого реализует интерфейс **Paint**. Этот объект передается методу **setPaint()**. В прикладном программном интерфейсе Java 2D API для этой цели предусмотрены следующие классы.

- Класс **Color** служит для заполнения фигуры сплошным цветом. Для этого достаточно вызвать метод **setPaint()**, указав объект типа **Color**:

```
g2.setPaint(Color.red);
```
- Класс **GradientPaint** служит для заполнения фигуры градиентом, т.е. цветом, постепенно изменяющим свой оттенок (рис. 11.15).
- Класс **TexturePaint** служит для заполнения фигуры текстурой, т.е. повторяющимся рисунком (рис. 11.16).

Для создания объекта типа **GradientPaint** следует указать две точки изменения градиента и цвета, которые должны сменять друг друга в этих точках, как показано ниже.

```
g2.setPaint(new GradientPaint(p1, Color.RED, p2, Color.YELLOW));
```

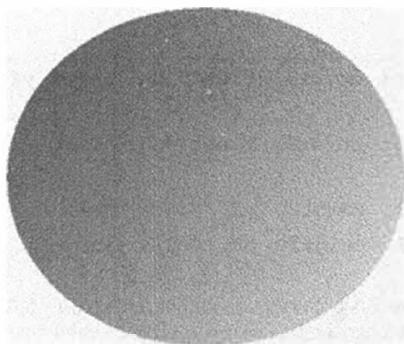


Рис. 11.15. Градиентная раскраска

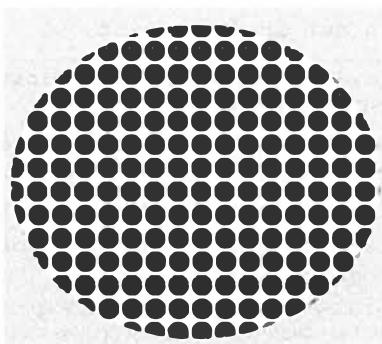


Рис. 11.16. Текстурная раскраска

Постепенное изменение цвета осуществляется по прямой, соединяющей эти точки. Цвета остаются постоянными вдоль прямых, перпендикулярных к линии соединения. Для точек, находящихся за пределами соединительной линии, задаются цвета ее конечных точек соответственно.

Если же вызывать конструктор класса `GradientPaint` с параметром `cyclic`, принимающим логическое значение `true`, то изменение цвета будет происходить циклически за пределами конечных точек соединительной линии.

```
g2.setPaint(new GradientPaint(p1, Color.RED, p2, Color.YELLOW, true));
```

Для создания объекта типа `TexturePaint` следует указать объект типа `BufferedImage` повторяющегося рисунка и *прямоугольник привязки*:

```
g2.setPaint(new TexturePaint(bufferedImage, anchorRectangle));
```

Класс `BufferedImage` будет более подробно рассматриваться далее в этой главе, когда дойдет очередь до обсуждения приемов обработки изображений. А до тех пор достаточно сказать, что самый простой способ получить буферизованное изображение в виде объекта типа `BufferedImage` — ввести его из файла, как показано ниже.

```
bufferedImage = ImageIO.read(new File("blue-ball.gif"));
```

Прямоугольник привязки повторяется бесконечное количество раз в направлениях, параллельных осям *x* и *y*, образуя текстуру в виде мозаики. Исходный рисунок масштабируется таким образом, чтобы вписаться в прямоугольник привязки, а затем повторяется в каждом элементе мозаики.

java.awt.Graphics2D 1.2

- **void setPaint(Paint s)**

Устанавливает в качестве раскраски для данного графического контекста указанный объект, класс которого реализует интерфейс `Paint`.

java.awt.GradientPaint 1.2

- **GradientPaint(float x1, float y1, Color color1, float x2, float y2, Color color2)**
- **GradientPaint(float x1, float y1, Color color1, float x2, float y2, Color color2, boolean cyclic)**
- **GradientPaint(Point2D p1, Color color1, Point2D p2, Color color2)**
- **GradientPaint(Point2D p1, Color color1, Point2D p2, Color color2, boolean cyclic)**

Создают объект градиентной раскраски. С его помощью фигура заполняется таким образом, чтобы начальная точка с координатами **x1** и **y1** была окрашена цветом **color1**, конечная точка с координатами **x2** и **y2** — цветом **color2**, а в промежутке между ними цвет изменялся линейно путем интерполяции. Цвета остаются постоянными вдоль прямой, перпендикулярной линии, соединяющей начальную и конечную точки. По умолчанию градиентная раскраска не выполняется циклически. Это означает, что точки, находящиеся за соответствующими пределами градиента, окрашиваются тем же самым цветом, что и начальная и конечная точки градиента. Если же градиентная раскраска выполняется циклически, то интерполяция цветов продолжается, возвращаясь сначала к цвету начальной точки, а затем непрерывно повторяясь в обоих направлениях.

java.awt.TexturePaint 1.2

- **TexturePaint(BufferedImage texture, Rectangle2D anchor)**

Создает объект текстурной раскраски. Прямоугольник привязки определяет мозаичное заполнение раскрашиваемого участка. Он повторяется бесконечное число раз в направлении обеих осей координат, **x** и **y**, масштабируя заданное изображение текстуры таким образом, чтобы оно заполняло каждый элемент мозаики.

11.6. Преобразование координат

Допустим, требуется нарисовать такой объект, как автомобиль, по известным исходным данным (высоте, расстоянию между осями и общей длине в метрах). Конечно, все координаты рисуемых частей автомобиля можно выразить в пикселях, подсчитав количество пикселей, приходящихся на метр. Но все это можно сделать намного проще, запросив соответствующий графический контекст и выполнив в нем преобразование координат следующим образом:

```
g2.scale(pixelsPerMeter, pixelsPerMeter);
g2.draw(new Line2D.Double(координаты_в_метрах)); // преобразовать в
// пиксели и нарисовать соответственно масштабированную линию
```

Метод **scale()** из класса **Graphics2D** выполняет масштабное преобразование координат в заданном графическом контексте. При этом пользовательские координаты (единицы измерения, указанные пользователем) преобразуются в *аппаратные координаты* (пиксели). На рис. 11.17 приведен пример такого преобразования.

Преобразование координат очень удобно применять на практике, поскольку оно позволяет оперировать общепринятыми единицами измерения. А все хлопоты по их преобразованию в пиксели берет на себя заданный графический контекст.



Рис. 11.17. Пользовательские и аппаратные координаты

В языке Java предусмотрено четыре основных вида преобразований координат.

- **Масштабирование.** Увеличение или сокращение всех расстояний от фиксированной точки.
- **Вращение.** Поворот всех точек фигуры вокруг фиксированной точки.
- **Перемещение.** Передвижение всех точек фигуры на фиксированное расстояние.
- **Сдвиг.** Фиксация одной линии и передвижение всех параллельных ей линий фигуры на расстояние, пропорциональное расстоянию от данной линии до фиксированной.

На рис. 11.18 показаны результаты перечисленных выше видов преобразования над единичным квадратом.

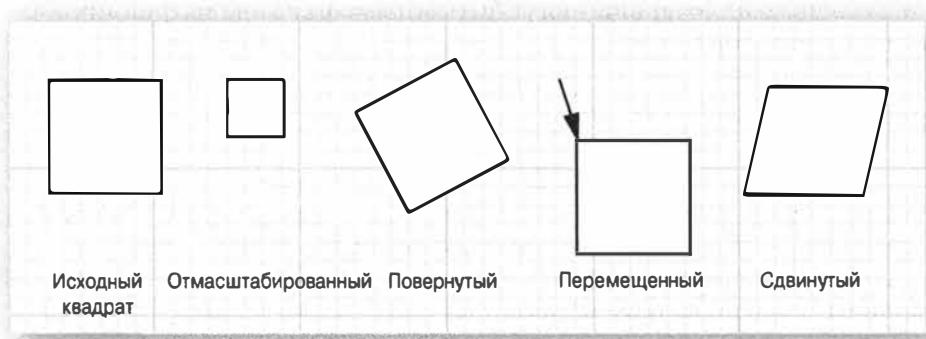


Рис. 11.18. Основные виды преобразования координат

Методы `scale()`, `rotate()`, `translate()` и `shear()` из класса `Graphics2D` реализуют перечисленные выше виды преобразования. Применяя эти методы в разных сочетаниях, можно выполнять *составные* преобразования, например, повернуть фигуру и удвоить ее размеры. Для этого достаточно выполнить операции вращения и масштабирования следующим образом:

```
g2.rotate(angle);
g2.scale(2, 2);
g2.draw(..);
```

В данном случае не важно, в какой именно последовательности будут выполняться преобразования. Но в целом порядок выполнения большинства преобразований имеет значение. Так, если нужно повернуть и сдвинуть изображение, сначала необходимо решить, какое именно преобразование выполнить первым, чтобы добиться желаемого результата. Преобразования в графическом контексте выполняются в обратном порядке по сравнению с тем, как они были указаны. Это означает, что преобразование, указанное последним, выполняется первым.

Допускается любое количество преобразований координат. Рассмотрим следующую последовательность преобразований:

```
g2.translate(x, y);
g2.rotate(a);
g2.translate(-x, -y);
```

Последнее преобразование (которое выполняется первым) перемещает всю фигуру таким образом, чтобы точка с координатами (x, y) совпала с точкой начала отсчета. Второе преобразование поворачивает фигуру на угол a вокруг точки начала отсчета. А последнее преобразование снова перемещает всю фигуру таким образом, чтобы точка с координатами (x, y) совпала с точкой начала отсчета. В итоге фигура поворачивается вокруг точки с координатами (x, y) , как показано на рис. 11.19. Вращение фигуры вокруг точки, отличающейся от начала отсчета, выполняется довольно часто, поэтому для выполнения данной операции предусмотрен следующий метод:

```
g2.rotate(a, x, y);
```

Как известно из теории матриц, вращение, масштабирование, перемещение, сдвиг и различные их сочетания могут быть выражены в виде матриц преобразования следующим образом:

$$\begin{bmatrix} x_{\text{new}} \\ y_{\text{new}} \\ 1 \end{bmatrix} = \begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Такие преобразования называются *аффинными*, а в прикладном программном интерфейсе Java 2D API они описываются с помощью класса `AffineTransform`. Объект аффинного преобразования можно создать непосредственно, если известны компоненты матрицы преобразования:

```
AffineTransform t = new AffineTransform(a, b, c, d, e, f);
```

Для реализации отдельных типов преобразований предусмотрены также фабричные методы `getRotateInstance()`, `getScaleInstance()`,

`getTranslateInstance()` и `getShearInstance()`, назначение которых можно легко определить по их названию. Например, при вызове метода
`t = AffineTransform.getScaleInstance(2.0F, 0.5F);`
возвращается преобразование, которое соответствует такой матрице:

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

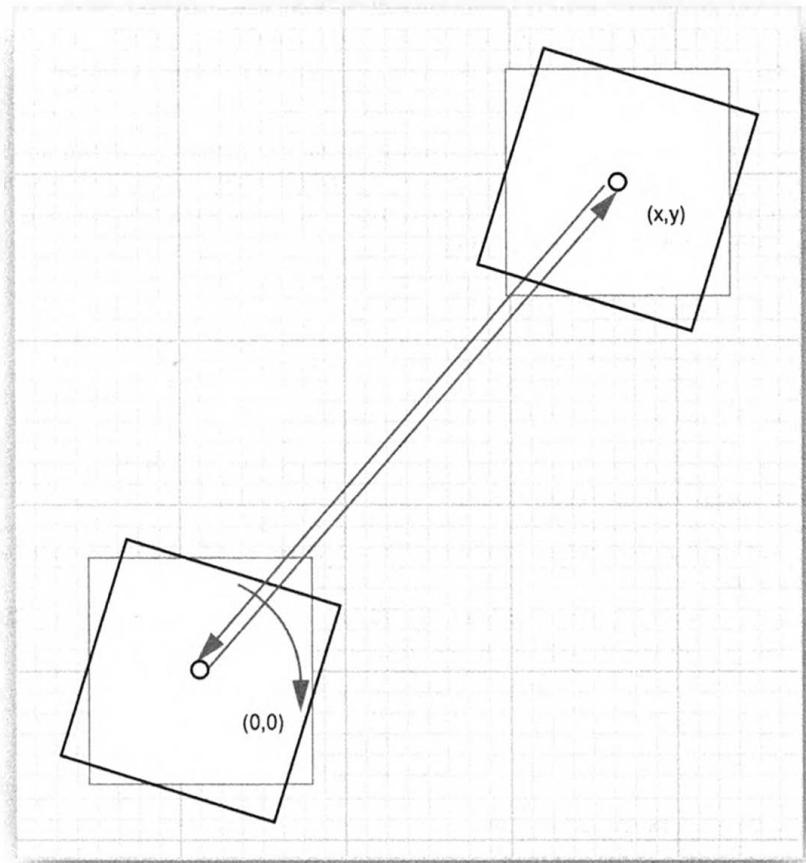


Рис. 11.19. Составные преобразования

И наконец, методы экземпляра `setToRotate()`, `setToScale()`, `setToTranslation()` и `setToShear()` служат для указания нового вида преобразования. Ниже приведен характерный тому пример.

```
t.setToRotation(angle); // установить вращение на заданный угол
```

Для замены текущего преобразования координат в графическом контексте аффинным преобразованием, представленным объектом типа `AffineTransform`, служит метод

```
g2.setTransform(t); // заменить текущее преобразование
```

Но на практике вызывать метод `setTransform()` все же не рекомендуется, поскольку он заменяет любое преобразование, которое может присутствовать в графическом контексте. Например, графический контекст для печати страницы с альбомной ориентацией уже содержит преобразование вращением на 90°. А при вызове метода `setTransform()` это преобразование игнорируется. В таком случае вместо метода `setTransform()` рекомендуется вызвать метод `transform()`, как показано ниже. Этот метод объединяет текущее преобразование с новым объектом типа `AffineTransform`, представляющим аффинное преобразование.

```
g2.transform(t); // составить текущее преобразование вместе с аффинным
```

Если же требуется временно выполнить какое-нибудь преобразование, то сначала следует сохранить предыдущее преобразование, затем составить вместе с ним новое и, наконец, восстановить прежнее преобразование, как показано в приведенном ниже фрагменте кода.

```
AffineTransform oldTransform = g2.getTransform();
    // сохранить прежнее преобразование
g2.transform(t); // выполнить временное преобразование
рисовать в графическом контексте g2
g2.setTransform(oldTransform);
    // восстановить прежнее преобразование
```

`java.awt.geom.AffineTransform 1.2`

- `AffineTransform(double a, double b, double c, double d, double e, double f)`
- `AffineTransform(float a, float b, float c, float d, float e, float f)`
Конструируют аффинное преобразование по приведенной ниже матрице.

$$\begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix}$$

- `AffineTransform(double[] m)`
- `AffineTransform(float[] m)`

Конструируют аффинное преобразование по приведенной ниже матрице.

$$\begin{bmatrix} m[0] & m[2] & m[4] \\ m[1] & m[3] & m[5] \\ 0 & 0 & 1 \end{bmatrix}$$

java.awt.geom.AffineTransform 1.2 (окончание)

- **static AffineTransform getRotateInstance(double a)**

Задает преобразование вращением против часовой стрелки вокруг точки начала отсчета на угол **a**, указываемый в радианах. Ниже приведена матрица такого преобразования.

$$\begin{bmatrix} \cos(a) & -\sin(a) & 0 \\ \sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Если значение параметра **a** находится в пределах от 0 до $\pi/2$, то вращение происходит в направлении от положительной оси **x** до положительной оси **y**.

- **static AffineTransform getRotateInstance(double a, double x, double y)**

Задает преобразование вращением против часовой стрелки вокруг точки с координатами **[x, y]** на угол **a**, указываемый в радианах.

- **static AffineTransform getScaleInstance(double sx, double sy)**

Задает преобразование масштабированием с масштабными коэффициентами **sx** и **sy** по осям **x** и **y** соответственно. Ниже приведена матрица такого преобразования.

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- **static AffineTransform getShearInstance (double shx, double shy)**

Задает преобразование сдвигом с коэффициентами **shx** и **shy** по осям **x** и **y**. Ниже приведена матрица такого преобразования.

$$\begin{bmatrix} 1 & shx & 0 \\ shy & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- **static AffineTransform getTranslateInstance(double tx, double ty)**

Задает преобразование перемещением на расстояния **tx** и **ty** по осям **x** и **y**. Ниже приведена матрица этого преобразования.

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}$$

- **void setToRotation(double a)**
- **void setToRotation(double a, double x, double y)**
- **void setScale(double sx, double sy)**
- **void setShear(double sx, double sy)**
- **void setTranslation(double tx, double ty)**

Устанавливают аффинное преобразование в соответствии с указанными параметрами. Эти параметры интерпретируются таким же образом, как и в упомянутых выше методах типа **getXXXInstance()** для основных преобразований.

java.awt.Graphics2D 1.2

- **void setTransform(AffineTransform t)**

Заменяет существующее преобразование координат в данном графическом контексте указанным аффинным преобразованием *t*.

- **void transform(AffineTransform t)**

Составляет преобразование координат, существующее в данном графическом контексте, вместе с аффинным преобразованием *t*.

- **void rotate(double a)**

- **void rotate(double a, double x, double y)**

- **void scale(double sx, double sy)**

- **void shear(double sx, double sy)**

- **void translate(double tx, double ty)**

Составляют преобразование координат, существующее в данном графическом контексте, вместе с основным преобразованием по указанным параметрам. Эти параметры интерпретируются таким же образом, как и в упомянутых выше методах типа **getXXXInstance()** для основных преобразований.

11.7. Отсечение

Для выполнения всех графических операций только в ограниченном участке графического контекста предусмотрена *фигура отсечения*:

```
g2.setClip(clipShape);
    // лучше воспользоваться приведенный ниже методом
g2.draw(shape); // рисовать только внутри фигуры отсечения
```

Но на практике вызывать метод **setClip()** не рекомендуется, поскольку он заменяет все существующие фигуры отсечения в данном графическом контексте. Как будет показано далее, графический контекст для печати уже содержит прямоугольник отсечения, который позволяет избежать появления данных на полях страницы. В этом случае вместо метода **setClip()** лучше вызвать метод **clip()** следующим образом:

```
g2.clip(clipShape); // лучше вызвать именно этот метод
```

Метод **clip()** образует пересечение существующей фигуры отсечения с указанной фигурой. Если же фигуру отсечения требуется применить лишь временно, то сначала следует сохранить прежнюю фигуру, затем добавить новую и восстановить прежнюю. Ниже приведен соответствующий пример.

```
Shape oldClip = g2.getClip(); // сохранить прежнее отсечение
g2.clip(clipShape); // осуществить временное отсечение
рисовать в графическом контексте g2
g2.setClip(oldClip); // восстановит прежнее отсечение
```

На рис. 11.20 возможности отсечения демонстрируются на примере рисования довольно непростого штрихового рисунка, который отсекается сложной фигурой, а именно контуром ряда символов.



Рис. 11.20. Отсечение штрихового рисунка фигурами букв

Для получения контуров символов требуется контекст воспроизведения шрифтов. С этой целью сначала вызывается метод `getFontRenderContext()` из класса `Graphics2D`:

```
FontRenderContext context = g2.getFontRenderContext();
```

Затем создается объект расположения текста типа `TextLayout`, для чего используется символьная строка, шрифт и контекст воспроизведения шрифтов:

```
TextLayout layout = new TextLayout("Hello", font, context);
```

Объект расположения текста описывает последовательность символов, расположение и оформление которых определяется выбранным контекстом воспроизведения шрифтов. Как известно, одни и те же символы могут по-разному выглядеть на экране и на печатной странице.

Но важнее другое: метод `getOutline()` возвращает объект типа `Shape`, описывающий фигуру контура символов в расположении текста. Эта фигура начинается в точке начала отсчета с координатами `(0, 0)`. Но если такое расположение не подходит, то при вызове метода `getOutline()` задается аффинное преобразование, позволяющее указать, в какой именно точке должен появиться контур:

```
AffineTransform transform =
    AffineTransform.getTranslateInstance(0, 100);
Shape outline = layout.getOutline(transform);
```

А затем контур присоединяется к фигуре отсечения следующим образом:

```
GeneralPath clipShape = new GeneralPath();
clipShape.append(outline, false);
```

И наконец, устанавливается фигура отсечения и рисуются линии штриховки, как показано ниже. В итоге линии появляются только внутри символов.

```
g2.setClip(clipShape);
Point2D p = new Point2D.Double(0, 0);
for (int i = 0; i < NLines; i++)
{
    double x = . . .;
    double y = . . .;
    Point2D q = new Point2D.Double(x, y);
    g2.draw(new Line2D.Double(p, q)); // отсечь линии штриховки
}
```

java.awt.Graphics 1.0

- **void setClip(Shape s) 1.2**
Задает фигуру *s* в качестве текущей фигуры отсечения.
- **Shape getClip() 1.2**
Возвращает текущую фигуру отсечения.

java.awt.Graphics2D 1.2

- **void clip(Shape s)**
Образует пересечение текущей фигуры отсечения с фигурой *s*.
- **FontRenderContext getFontRenderContext()**
Возвращает контекст воспроизведения шрифтов, требующийся для создания объекта типа *TextLayout*.

java.awt.font.TextLayout 1.2

- **TextLayout(String s, Font f, FontRenderContext context)**
Создает объект *TextLayout*, исходя из заданной символьной строки, шрифта и контекста воспроизведения шрифтов.
- **float getAdvance()**
Возвращает ширину данного расположения текста.
- **float getAscent()**
- **float getDescent()**
Возвращают высоту данного расположения текста относительно базовой линии.
- **float getLeading()**
Возвращает расстояние между соседними строками для шрифта, применяемого в данном расположении текста.

11.8. Прозрачность и композиция

В стандартной цветовой модели RGB каждый цвет описывается по трем его основным составляющим: красной, зеленой и синей. Но иногда отдельные участки изображения удобно сделать прозрачными или частично прозрачными. При наложении рисунка на уже существующее изображение прозрачные пиксели не закрывают находящиеся под ними пиксели, а частично прозрачные пиксели смешиваются с нижележащими пикселями. На рис. 11.21 приведен результат наложения частично прозрачного прямоугольника на уже имеющееся изображение. Обратите внимание на то, что детали изображения, находящиеся под прямоугольником, по-прежнему видны.

В прикладном программном интерфейсе Java 2D API прозрачность описывается с помощью *альфа-канала*. Каждый пиксель, кроме красной, зеленой и синей составляющей цвета, имеет значение прозрачности в альфа-канале, изменяющееся в предела от 0 (полностью прозрачен) до 1 (совершенно непрозрачен).

Например, прямоугольник на рис. 11.21 заполнен бледно-желтым цветом с прозрачностью 50% следующим образом:

```
new Color(0.7F, 0.7F, 0.0F, 0.5F);
```



Рис. 11.21. Наложение частично прозрачного прямоугольника на изображение

А теперь рассмотрим пример наложения двух фигур. Для этого требуется смешать или составить цвета и значения прозрачности в альфа-канале исходных и целевых пикселей. Исследователи Портер (Porter) и Дафф (Duff) в области компьютерной графики сформулировали двенадцать возможных правил композиции, которые реализованы в прикладном программном интерфейсе Java 2D API, однако только два из них имеют практическое применение. Если эти правила покажутся слишком сложными, то вместо них рекомендуется использовать простое правило SRC_OVER, которое применяется по умолчанию для объектов типа `Graphics2D` и дает интуитивно понятные результаты.

Рассмотрим вкратце теоретические основы правил композиции. Допустим, что *исходный пиксель* имеет значение прозрачности в альфа-канале a_s , а *целевой пиксель* — значение прозрачности a_d . На рис. 11.22 показана диаграмма, схематически поясняющая правила композиции для этих значений.

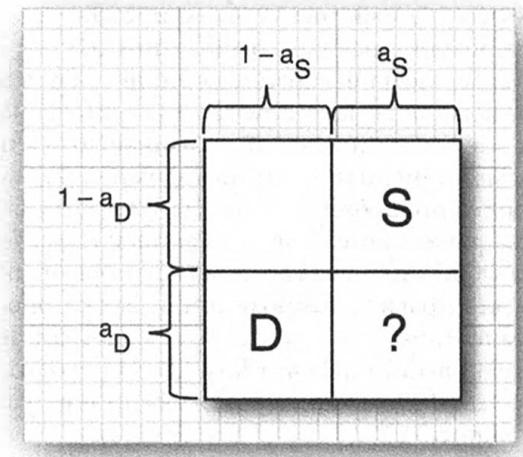


Рис. 11.22. Схематическое представление составляемого правила композиции

Портер и Дафф считают, что значение прозрачности в альфа-канале выражает вероятность использования цвета пикселя при объединении изображений. Для исходного пикселя первоначальный цвет будет использоваться с вероятностью a_s и не использоваться с вероятностью $1 - a_s$. Это же справедливо и для целевого пикселя. Допустим, что при составлении цветов эти вероятности независимы. На рис. 11.22 показаны четыре возможные ситуации. Если требуется использовать преимущественно цвет исходного, а не целевого пикселя (эта ситуация обозначена буквой S), то вероятность такого события равна $a_s \cdot (1 - a_d)$. Аналогично вычисляется вероятность $a_d \cdot (1 - a_s)$ преимущественного использования цвета целевого, а не исходного пикселя (эта ситуация обозначена буквой D). Что же делать, если в исходном и целевом изображениях преимущественно выбирается свой цвет? Именно в этой ситуации используются правила композиции Портера–Даффа. Если предпочтение отдается исходному цвету, в таком случае правый нижний угол диаграммы помечается буквой S, а само правило композиции называется SRC_OVER. По этому правилу исходный и целевой цвета сочетаются с весом a_s и $a_d \cdot (1 - a_s)$ соответственно.

Визуальный эффект применения этого правила композиции состоит в том, что при смешении цветов исходного и целевого пикселей приоритет отдается цвету исходного пикселя. В частности, если $a_s = 1$, то цвет целевого пикселя вообще не принимается во внимание. А если $a_s = 0$, то исходный пиксель становится совершенно прозрачным, тогда как цвет целевого пикселя не изменяется.

В зависимости от того, как расставлены буквы на диаграмме, можно сформировать другие правила композиции. Все правила композиции, поддерживаемые в прикладном программном интерфейсе Java 2D API, перечислены в табл. 11.1 и схематически показаны на рис. 11.23. Изображения на этом рисунке представляют результаты применения правил композиции к прямоугольному участку исходного изображения со значением прозрачности 0,75 в альфа-канале и эллиптическому участку целевого изображения со значением прозрачности 1,0 в альфа-канале.

Как можно заметить, большинство этих правил вряд ли применяются на практике. Так, например, правило композиции DST_IN представляет собой крайний случай, когда во внимание совсем не принимается цвет исходного пикселя, тогда как его альфа-канал используется для изменения целевого пикселя. В отличие от него, правило композиции SRC может оказаться удобным, потому что оно предписывает использовать исходный цвет, исключая смешение с целевым цветом. Более подробно о правилах Портера–Даффа можно прочитать в упоминавшейся ранее книге *Computer Graphics: Principles and Practice, Second Edition* in C Джеймса Фоли, Андриса ван Дама, Стивена Фэйнера и др.

Для создания объекта правила композиции, класс которого реализует интерфейс Composite, служит метод setComposite() из класса Graphics2D. В состав прикладного программного интерфейса Java 2D API входит только один такой класс — AlphaComposite. В этом классе реализованы все правила Портера–Даффа, представленные на рис. 11.23.

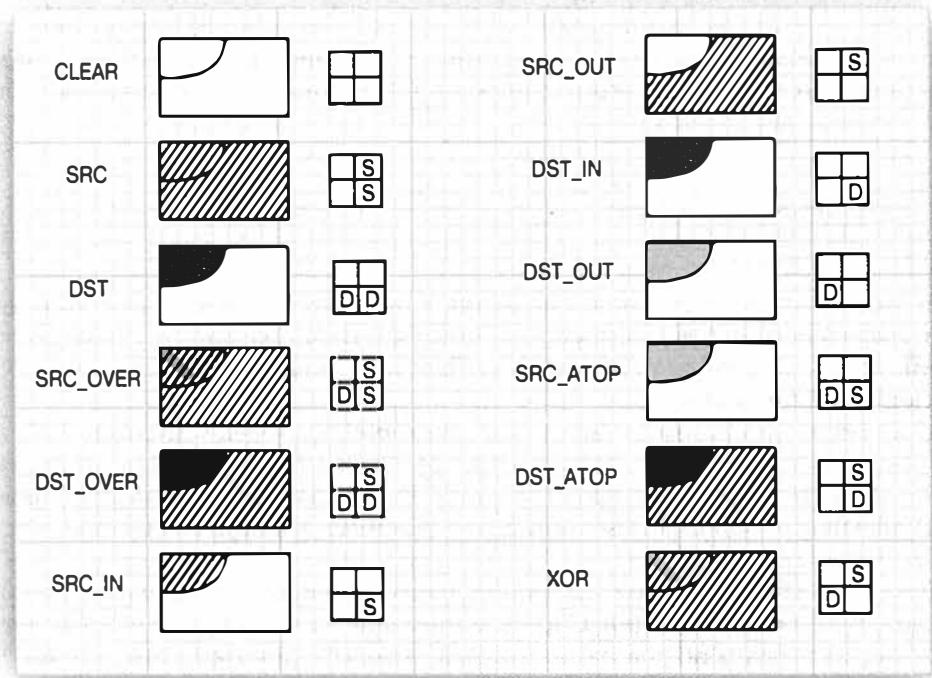


Рис. 11.23. Правила композиции Портера–Даффа

Таблица 11.1. Правила композиции Портера–Даффа

Правило	Описание
CLEAR	Исходные пиксели очищают целевые
SRC	Исходные пиксели перезаписывают целевые и пустые пиксели
DST	Исходные пиксели не оказывают никакого влияния на целевые пиксели
SRC_OVER	Исходные пиксели смешиваются с целевыми и перезаписывают пустые пиксели
DST_OVER	Исходные пиксели не оказывают никакого влияния на целевые пиксели и перезаписывают пустые пиксели
SRC_IN	Исходные пиксели перезаписывают целевые пиксели
SRC_OUT	Исходные пиксели очищают целевые пиксели и перезаписывают пустые пиксели
DST_IN	Прозрачность в альфа-канале исходного изображения видоизменяет целевое изображение
DST_OUT	Дополнение до прозрачности в альфа-канале исходного изображения видоизменяет целевое изображение
SRC_ATOP	Исходные пиксели смешиваются с целевыми
DST_ATOP	Прозрачность в альфа-канале исходного изображения видоизменяет целевое изображение. А исходные пиксели перезаписывают пустые пиксели
XOR	Дополнение до прозрачности в альфа-канале исходного изображения видоизменяет целевое изображение. А исходные пиксели перезаписывают пустые пиксели

Для создания экземпляра правила типа AlphaComposite служит фабричный метод `getInstance()` из класса `AlphaComposite`. В качестве параметров этого метода указываются правило композиции и значение прозрачности в альфа-канале для пикселей исходного изображения, как показано в приведенном ниже фрагменте кода.

```
int rule = AlphaComposite.SRC_OVER;
float alpha = 0.5f;
g2.setComposite(AlphaComposite.getInstance(rule, alpha));
g2.setPaint(Color.blue);
g2.fill(rectangle);
```

В этом фрагменте кода прямоугольник заполняется синим цветом со значением прозрачности 0,5 в альфа-канале. В соответствии с заданным правилом композиции `SRC_OVER` этот прямоугольник прозрачно накладывается на уже существующее изображение.

В примере программы, исходный код которой приведен в листинге 11.3, предоставляется возможность исследовать правила композиции. Для этого достаточно выбрать конкретное правило из комбинированного списка и установить ползунковым регулятором значение прозрачности в альфа-канале для объекта типа `AlphaComposite`.

В нижней части рабочего окна данной программы приводится краткое описание каждого правила композиции. Обратите внимание на то, что это описание автоматически составляется по диаграммам правил композиции. Например, строка "DS" во втором ряду диаграммы приводит к появлению в описании правила композиции строки "blends with destination" (смешивается с цветом целевого изображения).

Но эта программа не гарантирует, что используемый графический контекст, соответствующий экрану, имеет альфа-канал. Когда пиксели накладываются на целевое изображение без альфа-канала, цвета пикселей умножаются на значение прозрачности в альфа-канале, а затем это значение отбрасывается. В ряде правил композиции Портера-Даффа используются значения прозрачности в альфа-канале целевого изображения, и это указывает на важную роль альфа-канала. Поэтому для составления изображений (в данном случае фигур) используется буферизованное изображение с цветовой моделью ARGB. После составления результирующее изображение выводится на экран, как показано в приведенном ниже фрагменте кода.

```
BufferedImage image = new BufferedImage(getWidth(), getHeight(),
    BufferedImage.TYPE_INT_ARGB);
Graphics2D gImage = image.createGraphics();
// а теперь рисовать на изображении gImage
g2.drawImage(image, null, 0, 0);
```

В листингах 11.3 и 11.4 представлен исходный код классов фрейма и компонента. А класс `Rule` из листинга 11.5 предоставляет краткое описание каждого правила композиции, как показано на рис. 11.24. После запуска программы на выполнение переместите ползунок регулятора слева направо, чтобы посмотреть результат применения выбранного правила композиции для наложения фигур при разных значениях прозрачности в альфа-канале. Обратите особое внимание на то, что правила `DST_IN` и `DST_OUT` отличаются лишь направлением изменения цвета целевого изображения (!) при изменении значения прозрачности в альфа-канале исходного изображения.

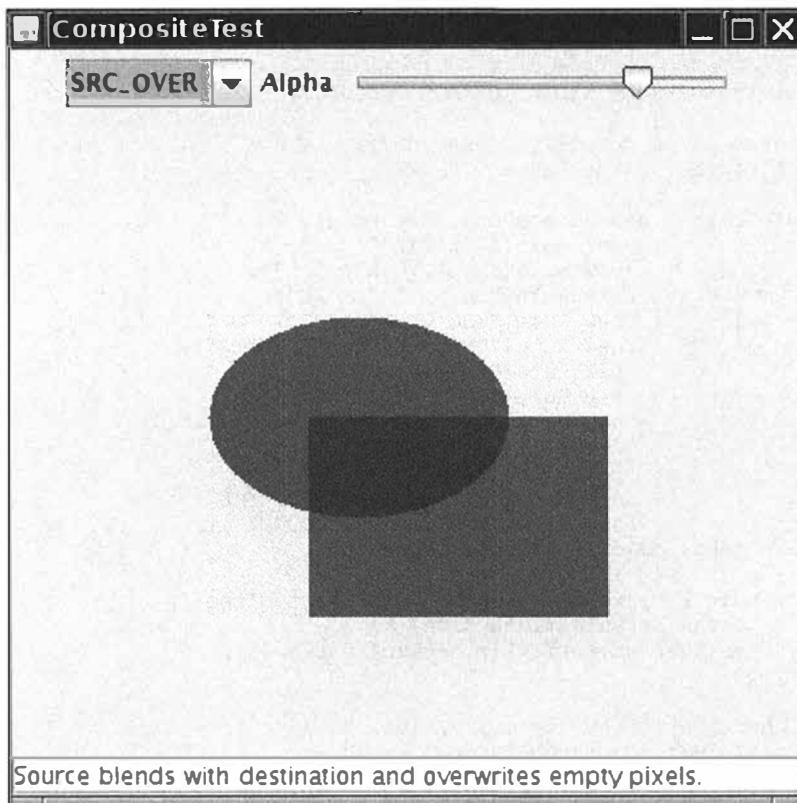


Рис. 11.24. Рабочее окно программы CompositeTest

Листинг 11.3. Исходный код из файла composite/CompositeTestFrame.java

```
1 package composite;
2
3 import java.awt.*;
4
5 import javax.swing.*;
6
7 /**
8  * Этот фрейм содержит комбинированный список для выбора правил,
9  * композиции, ползунок для изменения прозрачности в альфа-канале
10 * исходного изображения, а также компонент для отображения
11 * результатов составления изображений
12 */
13 class CompositeTestFrame extends JFrame
14 {
15     private static final int DEFAULT_WIDTH = 400;
16     private static final int DEFAULT_HEIGHT = 400;
17
18     private CompositeComponent canvas;
19     private JComboBox<Rule> ruleCombo;
20     private JSlider alphaSlider;
21     private JTextField explanation;
```

```

22
23     public CompositeTestFrame()
24     {
25         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
26
27         canvas = new CompositeComponent();
28         add(canvas, BorderLayout.CENTER);
29
30         ruleCombo = new JComboBox<>(new Rule[]
31             { new Rule("CLEAR", " ", " "),
32               new Rule("SRC", " S", " S"),
33               new Rule("DST", " ", "DD"),
34               new Rule("SRC OVER", " S", "DS"),
35               new Rule("DST OVER", " S", "DD"),
36               new Rule("SRC IN", " ", " S"),
37               new Rule("SRC OUT", " S", " "),
38               new Rule("DST IN", " ", " D"),
39               new Rule("DST OUT", " ", "D"),
40               new Rule("SRC_ATOP", " ", "DS"),
41               new Rule("DST_ATOP", " S", " D"),
42               new Rule("XOR", " S", "D ")},));
43
44         ruleCombo.addActionListener(event ->
45         {
46             Rule r = (Rule) ruleCombo.getSelectedItem();
47             canvas.setRule(r.getValue());
48             explanation.setText(r.getExplanation());
49         });
50
51         alphaSlider = new JSlider(0, 100, 75);
52         alphaSlider.addChangeListener(event ->
53             canvas.setAlpha(alphaSlider.getValue()));
54         JPanel panel = new JPanel();
55         panel.add(ruleCombo);
56         panel.add(new JLabel("Alpha"));
57         panel.add(alphaSlider);
58         add(panel, BorderLayout.NORTH);
59
60         explanation = new JTextField();
61         add(explanation, BorderLayout.SOUTH);
62
63         canvas.setAlpha(alphaSlider.getValue());
64         Rule r = ruleCombo.getItemAt(ruleCombo.getSelectedIndex());
65         canvas.setRule(r.getValue());
66         explanation.setText(r.getExplanation());
67     }
68 }
```

Листинг 11.4. Исходный код из файла composite/CompositeComponent.java

```

1 package composite;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import java.awt.image.*;
6 import javax.swing.*;
7
8 /**
```

```
9  * Этот компонент рисует две формы,
10 * составленные по правилу композиции
11 */
12 class CompositeComponent extends JComponent
13 {
14     private int rule;
15     private Shape shape1;
16     private Shape shape2;
17     private float alpha;
18
19     public CompositeComponent()
20     {
21         shape1 = new Ellipse2D.Double(100, 100, 150, 100);
22         shape2 = new Rectangle2D.Double(150, 150, 150, 100);
23     }
24
25     public void paintComponent(Graphics g)
26     {
27         Graphics2D g2 = (Graphics2D) g;
28
29         BufferedImage image = new BufferedImage(getWidth(),
30                                         getHeight(), BufferedImage.TYPE_INT_ARGB);
31         Graphics2D gImage = image.createGraphics();
32         gImage.setPaint(Color.red);
33         gImage.fill(shape1);
34         AlphaComposite composite = AlphaComposite
35             .getInstance(rule, alpha);
36         gImage.setComposite(composite);
37         gImage.setPaint(Color.blue);
38         gImage.fill(shape2);
39         g2.drawImage(image, null, 0, 0);
40     }
41
42 /**
43 * Устанавливает правило композиции
44 * @param r Правило композиции (в виде константы
45 *          из класса AlphaComposite)
46 */
47 public void setRule(int r)
48 {
49     rule = r;
50     repaint();
51 }
52
53 /**
54 * Устанавливает значение прозрачности в
55 * альфа-канале исходного изображения
56 * @param a Значение прозрачности в пределах от 0 до 100
57 */
58 public void setAlpha(int a)
59 {
60     alpha = (float) a / 100.0F;
61     repaint();
62 }
63 }
```

Листинг 11.5. Исходный код из файла composite/Rule.java

```

1 package composite;
2
3 import java.awt.*;
4
5 /**
6  * Этот класс описывает правило композиции Портера-Даффа
7 */
8 class Rule
9 {
10    private String name;
11    private String porterDuff1;
12    private String porterDuff2;
13
14    /**
15     * Составляет правило композиции Портера-Даффа
16     * @param n Название правила композиции
17     * @param pd1 Первый ряд квадратов в правиле
18     *   композиции Портера-Дафа
19     * @param pd2 Второй ряд в правиле композиции Портера-Дафа
20     */
21    public Rule(String n, String pd1, String pd2)
22    {
23        name = n;
24        porterDuff1 = pd1;
25        porterDuff2 = pd2;
26    }
27
28    /**
29     * Получает объяснение композиции по данному правилу
30     * @return Возвращает объяснение композиции по данному правилу
31     */
32    public String getExplanation()
33    {
34        StringBuilder r = new StringBuilder("Source ");
35        if (porterDuff2.equals(" ")) r.append("clears");
36        if (porterDuff2.equals(" S")) r.append("overwrites");
37        if (porterDuff2.equals("DS")) r.append("blends with");
38        if (porterDuff2.equals(" D")) r.append("alpha modifies");
39        if (porterDuff2.equals("D "))
40            r.append("alpha complement modifies");
41        if (porterDuff2.equals("DD")) r.append("does not affect");
42        r.append(" destination");
43        if (porterDuff1.equals(" S"))
44            r.append(" and overwrites empty pixels");
45        r.append(".");
46        return r.toString();
47    }
48
49    public String toString()
50    {
51        return name;
52    }
53
54    /**
55     * Получает значение по данному правилу в классе AlphaComposite
56     * @return Возвращает значение константы из класса

```

```

57     *          AlphaComposite или значение -1, если
58     *          соответствующая константа отсутствует
59     */
60     public int getValue()
61     {
62         try
63         {
64             return (Integer)
65                 AlphaComposite.class.getField(name).get(null);
66         }
67         catch (Exception e)
68         {
69             return -1;
70         }
71     }
72 }
```

java.awt.Graphics2D 1.2

- **void setComposite(Composite s)**

Устанавливает указанный объект, класс которого реализует интерфейс **Composite**, в качестве правила композиции для данного графического контекста.

java.awt.AlphaComposite 1.2

- **static AlphaComposite getInstance(int rule)**

- **static AlphaComposite getInstance(int rule, float sourceAlpha)**

Создают объект композиции на основе заданного правила и значения в альфа-канале. Параметр **rule**, задающий правило композиции, может принимать одно из следующих значений: **CLEAR**, **DST**, **SRC**, **SRC_OVER**, **DST_OVER**, **SRC_ATOP**, **SRC_IN**, **SRC_OUT**, **DST_ATOP**, **DST_IN**, **DST_OUT**, **XOR**.

11.9. Указания по воспроизведению

Как следует из предыдущих разделов, процесс воспроизведения графики довольно сложный. И хотя прикладной программный интерфейс Java 2D API чаще всего действует необычайно быстро, иногда возникает потребность в контроле над достижением компромисса между быстродействием и качеством. Получить такой контроль можно, задав так называемые указания по воспроизведению. В частности, метод **setRenderingHint()** из класса **Graphics2D** позволяет задать лишь одно такое указание. А ключи и значения указаний по воспроизведению объявляются в классе **RenderingHints**. В табл. 11.2 вкратце перечислены возможные варианты выбора ключей и значений. Значения, оканчивающиеся на **_DEFAULT**, обозначают выбираемые по умолчанию варианты, обеспечивающие в конкретной реализации достижение наилучшего компромисса между производительностью и качеством.

Таблица 11.2. Указания по воспроизведению

Ключ	Значение	Описание
KEY_ANTIALIASING	VALUE_ANTIALIAS_ON VALUE_ANTIALIAS_OFF VALUE_ANTIALIAS_DEFAULT	Включение/отключение слаживания фигур
KEY_TEXT_ANTIALIASING	VALUE_TEXT_ANTIALIAS_ON VALUE_TEXT_ANTIALIAS_OFF VALUE_TEXT_ANTIALIAS_DEFAULT VALUE_TEXT_ANTIALIAS_GASP 6 VALUE_TEXT_ANTIALIAS_LCD_HRGB 6 VALUE_TEXT_ANTIALIAS_LCD_HBGR 6 VALUE_TEXT_ANTIALIAS_LCD_VRGB 6 VALUE_TEXT_ANTIALIAS_LCD_VBGR 6	Включение/отключение слаживания шрифтов. Значение VALUE_TEXT_ANTIALIAS_GASP подразумевает просмотр gasp-таблицы шрифта, чтобы решить, следует ли слаживать шрифт конкретного размера. А значения с префиксом LCD вызывают принудительное воспроизведение подпикселей на устройствах отображения конкретного типа (в данном случае жидкокристаллических)
KEY_FRACTIONALMETRICS	VALUE_FRACTIONALMETRICS_ON VALUE_FRACTIONALMETRICS_OFF VALUE_FRACTIONALMETRICS_DEFAULT	Включение/отключение вычисления дробных размеров шрифтов. Дробные размеры шрифтов позволяют улучшить расположение выделяемых ими символов
KEY_RENDERING	VALUE_RENDER_QUALITY VALUE_RENDER_SPEED VALUE_RENDER_DEFAULT	Выбор алгоритма рисования для достижения лучшего качества или большей скорости
KEY_STROKE_CONTROL 1..3	VALUE_STROKE_NORMALIZE VALUE_STROKE_PURE VALUE_STROKE_DEFAULT	Выбор между возможностью передать контроль над расположением линий обводки графическому ускорителю (что чревато их смещением на целую половину пикселя) и вычислять их расположение строго по правилу, которое требует, чтобы линии обводки проходили по центру пикселей
KEY_DITHERING	VALUE_DITHER_ENABLE VALUE_DITHER_DISABLE VALUE_DITHER_DEFAULT	Включение/отключение имитации полутона. Алгоритм имитации полутона основан на специальном распределении значений цвета расположенных рядом пикселей. (Однако нужно иметь в виду, что слаживание может мешать имитации полутона.)
KEY_ALPHA_INTERPOLATION	VALUE_ALPHA_INTERPOLATION_QUALITY VALUE_ALPHA_INTERPOLATION_SPEED VALUE_ALPHA_INTERPOLATION_DEFAULT	Включение/отключение точного вычисления при составлении изображений по альфа-каналу
KEY_COLOR_RENDERING	VALUE_COLOR_RENDER_QUALITY VALUE_COLOR_RENDER_SPEED VALUE_COLOR_RENDER_DEFAULT	Выбор между качеством или быстрым действием при воспроизведении цветов. Такой выбор приходится делать только в том случае, если используются разные цветовые пространства
KEY_INTERPOLATION	VALUE_INTERPOLATION_NEAREST_NEIGHBOR VALUE_INTERPOLATION_BILINEAR VALUE_INTERPOLATION_BICUBIC	Выбор правила интерполяции пикселей при масштабировании или вращении изображений

Наиболее полезным из указаний по воспроизведению является сглаживание, подразумевающее удаление всевозможных неровностей при рисовании различных линий. Как показано на рис. 11.25, наклонная линия рисуется в виде лестницы из пикселей, расположенных уступом. Иногда, особенно на мониторах с низким разрешением, она выглядит совершенно неприемлемо. Но если каждый пиксель рисуемой линии воспроизвести не только двумя контрастными цветами (например, черным и белым), а полутонаами, значения которых пропорциональны занимаемой площади пикселя, то такая линия будет выглядеть намного более гладкой. Такой алгоритм называется *сглаживанием*. Безусловно, он требует больше времени, расходуемого на расчет значений всех переходных цветов.

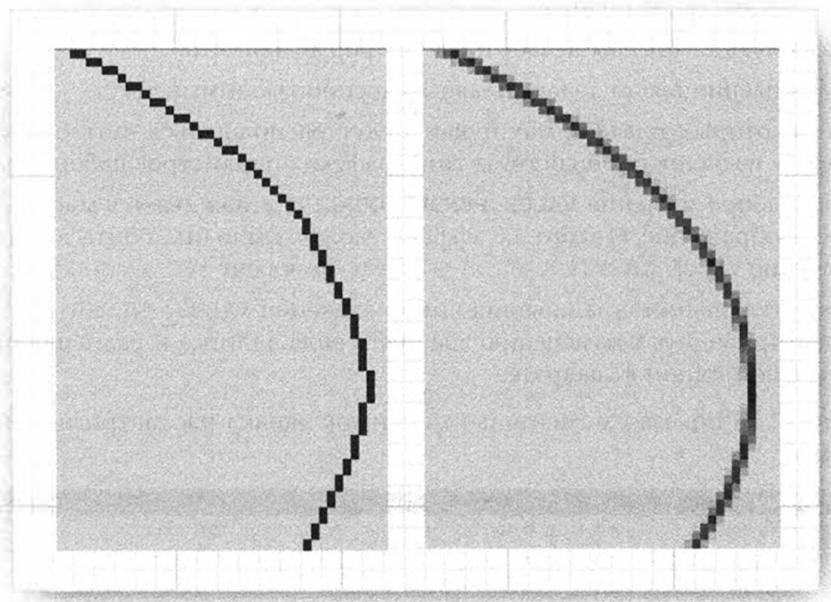


Рис. 11.25. Сглаживание

В качестве примера ниже показано, каким образом можно указать в коде на применение сглаживания.

```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,  
                    RenderingHints.VALUE_ANTIALIAS_ON);
```

Сглаживание целесообразно применять и для шрифтов, как показано ниже. Что же касается остальных указаний по воспроизведению, то они применяются реже.

```
g2.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,  
                    RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
```

Пары “ключ–значение” для указаний по воспроизведению можно собрать в единое отображение и задавать их одновременно с помощью метода `setRenderingHints()`. Для этого подойдет любой класс коллекций, реализующий интерфейс `Map`, но может вполне пригодиться и сам класс `RenderingHints`. Ведь он также реализует интерфейс `Map` и формирует отображение по умолчанию,

если параметр его конструктора принимает пустое значение null, как показано в приведенном ниже фрагменте кода.

```
RenderingHints hints = new RenderingHints(null);
hints.put(RenderingHints.KEY_ANTIALIASING,
          RenderingHints.VALUE_ANTIALIAS_ON);
hints.put(RenderingHints.KEY_TEXT_ANTIALIASING,
          RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
g2.setRenderingHints(hints);
```

Именно такой прием применяется в примере программы из листинга 11.6. В этой программе демонстрируются несколько указаний по воспроизведению, которые, на наш взгляд, наиболее полезны. В этой связи необходимо обратить особое внимание на следующее.

- Сглаживание фигуры делает плавной форму эллипса.
- Сглаживание текста делает плавным воспроизводимый текст.
- На некоторых платформах буквы будут располагаться чуть ближе друг к другу из-за дробной системы типографских параметров набора текста.
- При выборе значения `VALUE_RENDER_QUALITY` сглаживается масштабируемое изображение. (Такого же эффекта можно было бы добиться, установив значение `VALUE_INTERPOLATION_BICUBIC` по ключу `KEY_INTERPOLATION`.)
- При отключении сглаживания выбор значения `VALUE_STROKE_NORMALIZE` будет приводить к изменению внешнего вида эллипса и размещению диагональной линии в квадрате.

На рис. 11.26 показан моментальный снимок экрана рассматриваемой здесь программы.

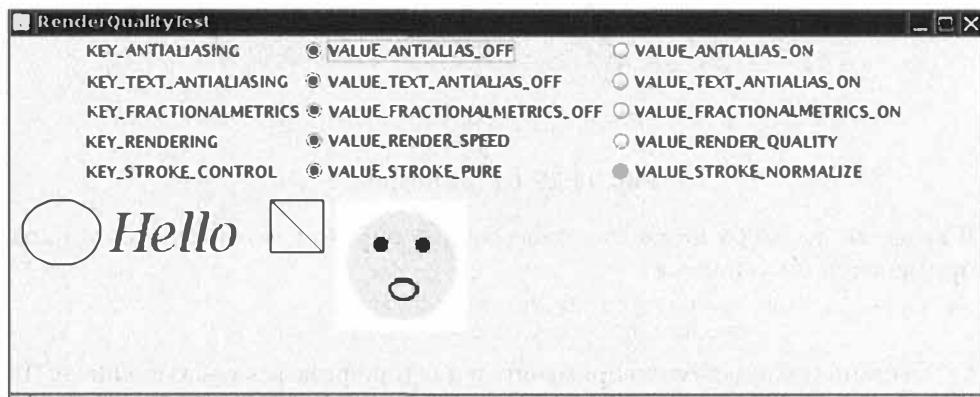


Рис. 11.26. Проверка действия указаний по воспроизведению

Листинг 11.6. Исходный код из файла renderQuality/RenderQualityTestFrame.java

```
1 package renderQuality;
2
3 import java.awt.*;
```

```
4 import java.awt.geom.*;
5
6 import javax.swing.*;
7
8 /**
9  * Этот фрейм содержит кнопки-переключатели для выбора указаний
10 * по воспроизведению, а также воспроизводимое по ним изображение
11 */
12 public class RenderQualityTestFrame extends JFrame
13 {
14     private RenderQualityComponent canvas;
15     private JPanel buttonBox;
16     private RenderingHints hints;
17     private int r;
18
19     public RenderQualityTestFrame()
20     {
21         buttonBox = new JPanel();
22         buttonBox.setLayout(new GridBagLayout());
23         hints = new RenderingHints(null);
24
25         makeButtons("KEY_ANTIALIASING", "VALUE_ANTIALIAS_OFF",
26                     "VALUE_ANTIALIAS_ON");
27         makeButtons("KEY_TEXT_ANTIALIASING",
28                     "VALUE_TEXT_ANTIALIAS_OFF",
29                     "VALUE_TEXT_ANTIALIAS_ON");
30         makeButtons("KEY_FRACTIONALMETRICS",
31                     "VALUE_FRACTIONALMETRICS_OFF",
32                     "VALUE_FRACTIONALMETRICS_ON");
33         makeButtons("KEY_RENDERING", "VALUE_RENDER_SPEED",
34                     "VALUE_RENDER_QUALITY");
35         makeButtons("KEY_STROKE_CONTROL", "VALUE_STROKE_PURE",
36                     "VALUE_STROKE_NORMALIZE");
37         canvas = new RenderQualityComponent();
38         canvas.setRenderingHints(hints);
39
40         add(canvas, BorderLayout.CENTER);
41         add(buttonBox, BorderLayout.NORTH);
42         pack();
43     }
44
45 /**
46  * Создает ряд кнопок-переключателей для выбора ключей и
47  * и значений конкретных указаний по воспроизведению
48  * @param key Имя ключа
49  * @param value1 Имя первого значения по данному ключу
50  * @param value2 Имя второго значения по данному ключу
51 */
52 void makeButtons(String key, String value1, String value2)
53 {
54     try
55     {
56         final RenderingHints.Key k = (RenderingHints.Key)
57             RenderingHints.class.getField(key).get(null);
58         final Object v1 =
59             RenderingHints.class.getField(value1).get(null);
60         final Object v2 =
61             RenderingHints.class.getField(value2).get(null);
62     }
```

```
62     JLabel label = new JLabel(key);
63
64     buttonBox.add(label, new GBC(0, r).setAnchor(GBC.WEST));
65     ButtonGroup group = new ButtonGroup();
66     JRadioButton b1 = new JRadioButton(value1, true);
67
68     buttonBox.add(b1, new GBC(1, r).setAnchor(GBC.WEST));
69     group.add(b1);
70     b1.addActionListener(event ->
71     {
72         hints.put(k, v1);
73         canvas.setRenderingHints(hints);
74     });
75     JRadioButton b2 = new JRadioButton(value2, false);
76
77     buttonBox.add(b2, new GBC(2, r).setAnchor(GBC.WEST));
78     group.add(b2);
79     b2.addActionListener(event ->
80     {
81         hints.put(k, v2);
82         canvas.setRenderingHints(hints);
83     });
84     hints.put(k, v1);
85     r++;
86 }
87 catch (Exception e)
88 {
89     e.printStackTrace();
90 }
91 }
92 }
93
94 /**
95 * Этот компонент производит рисунок, наглядно показывающий
96 * действие различных указаний по воспроизведению
97 */
98 class RenderQualityComponent extends JComponent
99 {
100     private static final Dimension PREFERRED_SIZE =
101             new Dimension(750, 150);
102     private RenderingHints hints = new RenderingHints(null);
103     private Image image;
104
105    public RenderQualityComponent()
106    {
107        image = new ImageIcon(getClass()
108                            .getResource("face.gif")).getImage();
109    }
110
111    public void paintComponent(Graphics g)
112    {
113        Graphics2D g2 = (Graphics2D) g;
114        g2.setRenderingHints(hints);
115
116        g2.draw(new Ellipse2D.Double(10, 10, 60, 50));
117        g2.setFont(new Font("Serif", Font.ITALIC, 40));
118        g2.drawString("Hello", 75, 50);
119    }
```

```

120     g2.draw(new Rectangle2D.Double(200, 10, 40, 40));
121     g2.draw(new Line2D.Double(201, 11, 239, 49));
122
123     g2.drawImage(image, 250, 10, 100, 100, null);
124 }
125
126 /**
127 * устанавливает указания по воспроизведению и
128 * выполняет перерисовку
129 * @param h Указания по воспроизведению
130 */
131 public void setRenderingHints(RenderingHints h)
132 {
133     hints = h;
134     repaint();
135 }
136
137 public Dimension getPreferredSize() { return PREFERRED_SIZE; }
138 }
```

java.awt.Graphics2D 1.2

- **void setRenderingHint(RenderingHints.Key key, Object value)**
Устанавливает указания по воспроизведению для данного графического контекста.
- **void setRenderingHints(Map m)**
Устанавливает все указания по воспроизведению, пары "ключ-значение" которых хранятся в заданном отображении.

java.awt.RenderingHints 1.2

- **RenderingHints (Map<RenderingHints.Key, ?> m)**
Создает отображение для хранения указаний по воспроизведению. Если параметр **m** принимает пустое значение **null**, то предоставляется отображение, реализуемое по умолчанию.

11.10. Чтение и запись изображений

В пакете `javax.imageio` содержатся готовые средства для чтения и записи файлов изображений в ряде наиболее употребительных форматов, а также библиотека для чтения и записи файлов других форматов. В частности, поддерживаются форматы GIF, JPEG, PNG, BMP (растровый формат для Windows) и WBMP (Wireless Bitmap — растровый формат для беспроводных сетей).

Основные функциональные возможности, доступные в библиотеке для чтения и записи изображений, чрезвычайно просты. Так, для загрузки изображения из файла применяется статический метод `read()` из класса `ImageIO`:

```
File f = . . . ;
BufferedImage image = ImageIO.read(f);
```

Класс `ImageIO` выбирает соответствующее средство чтения, исходя из типа файла. Он проверяет расширение файла и соответствующее значение в заголовке файла. Если для чтения данного файла нельзя найти подходящее средство или оно не в состоянии расшифровать содержимое файла, то статический метод `read()` из этого класса возвращает пустое значение `null`.

Так же просто осуществляется запись изображения в файл, как показано в приведенном ниже фрагменте кода. В данном случае форматирующая строка в переменной `format` определяет формат изображения (например, JPEG или PNG), а класс `ImageIO` выбирает соответствующее средство записи и сохраняет изображение в файле.

```
File f = . . . ;
String format = . . . ;
ImageIO.write(image, format, f);
```

11.10.1. Получение средств чтения и записи изображений по типам файлов

Для выполнения расширенных операций записи и чтения изображений, которые выходят за пределы простого использования статических методов `read()` и `write()` из класса `ImageIO`, необходимо прежде всего получить объекты типа `ImageReader` и `ImageWriter` соответственно. В классе `ImageIO` перечисляются средства чтения и записи, отвечающие одному из следующих условий.

- Формат изображения, например JPEG.
- Расширение файла, например `jpg`.
- Тип MIME, например `image/jpeg`.



На заметку! Стандарт MIME (Multipurpose Internet Mail Extensions — многоцелевые расширения почты в Интернете) определяет общие форматы данных (например, `image/jpeg` или `application/pdf`).

Например, с помощью приведенного ниже фрагмента кода можно получить средство чтения файлов изображений формата JPEG. Методы `getImageReaderBySuffix()` и `getImageReaderByMIMEType()` возвращают средства чтения файлов изображений с указанным расширением или типом MIME.

```
ImageReader reader = null;
Iterator<ImageReader> iter =
    ImageIO.getImageReadersByFormatName("JPEG");
if (iter.hasNext()) reader = iter.next();
```

Класс `ImageIO` позволяет обнаружить несколько средств чтения, каждое из которых способно обработать файлы изображений конкретного типа. В этом случае средство чтения выбирается исходя из более подробных сведений, которые можно получить с помощью интерфейса *поставщика услуг* следующим образом:

```
ImageReaderSpi spi = reader.getOriginatingProvider();
```

Затем можно получить название поставщика и номер версии:

```
String vendor = spi.getVendor();
String version = spi.getVersion();
```

Возможно, эти сведения помогут сделать выбор подходящего средства чтения или хотя бы составить список доступных средств, чтобы пользователи могли выбрать наиболее подходящее из них по своему усмотрению. Но для начала будем считать, что для чтения файлов изображений подходит первое же перечисленное средство.

В примере программы из листинга 11.7 требуется найти расширения файлов для всех доступных средств чтения, чтобы использовать их в фильтре файлов. Для этой цели следует вызвать статический метод `ImageIO.getReaderFileSuffixes()`, как показано ниже.

```
String[] extensions = ImageIO.getReaderFileSuffixes();
chooser.setFileFilter(new FileNameExtensionFilter(
    "Image files", extensions));
```

Что касается сохранения изображений в файлах, то для этого потребует-ся больше усилий. Хотелось бы, конечно, предоставить пользователю меню со всеми поддерживаемыми форматами изображений. Но, к сожалению, метод `getWriterFormNames()` из класса `IOImage` для этого не подходит, потому что он возвращает не совсем обычный перечень с лишними вариантами названий форматов, например, следующий:

```
jpg, BMP, bmp, JPEG, jpeg, wbmp, png, JPEG, PNG, WBMP, GIF, gif
```

Но это не совсем то, что должно отображаться в предполагаемом меню. Ведь это должен быть перечень только предпочтительных названий форматов. Поэтому для этой цели используется вспомогательный метод `getWriterFormats()` (см. листинг 11.7). Сначала в этом методе отыскивается первое средство записи, ассоциируемое с названием каждого формата, а затем у него запрашиваются имеющиеся названия форматов в надежде, что первым будет перечислен наиболее подходящий формат. Для средства записи изображений в файлы формата JPEG такой подход вполне пригоден, поскольку первым это средство, естественно, перечислит формат JPEG. (А вот средство записи изображений в файлы формата PNG перечислит первым не название формата PNG, а его строчный эквивалент `png`. Можно надеяться, что в ближайшем будущем этот недостаток будет устранен, а до тех пор все строчные буквы в названии форматов придется преобразовать в прописные.) После выбора названия предпочтительного формата все его альтернативные названия удаляются из исходного набора, и так происходит до тех пор, пока не будут обработаны все названия форматов.

11.10.2. Чтение и запись файлов с несколькими изображениями

Некоторые файлы, например анимационного формата GIF, могут содержать несколько изображений. Но статический метод `read()` из класса `ImageIO` позволяет прочитать только одно из них. Для чтения нескольких изображений необходимо преобразовать сначала источник входных данных (например, поток ввода или файл) в объект потока ввода изображений, относящийся к типу `ImageInputStream`, следующим образом:

```
InputStream in = . . . ;
ImageInputStream imageIn = ImageIO.createImageInputStream(in);
```

Затем этот объект следует присоединить к средству чтения, вызвав следующий метод:

```
reader.setInput(imageIn, true);
```

Второй параметр этого метода принимает логическое значение `true`, а это означает, что поток ввода находится в режиме просмотра данных только в прямом направлении. Логическое значение `false` этого параметра допускает произвольный доступ. В этом случае данные, читаемые из потока ввода, буферизуются или вводятся в режиме произвольного доступа к файлу. Произвольный доступ требуется только для определенных операций. Например, для подсчета количества изображений в файле анимационного формата GIF нужно прочитать весь этот файл. Если затем потребуется извлечь какое-нибудь изображение, то придется снова прочитать все введенные данные.

Это имеет значение только для чтения из потока ввода, если файл содержит несколько изображений, а формат изображения в его заголовке не предоставляет нужных сведений, например, о количестве изображений. Для чтения изображений непосредственно из файла можно воспользоваться приведенным ниже фрагментом кода.

```
File f = ...;
ImageInputStream imageIn = ImageIO.createImageInputStream(f);
reader.setInput(imageIn);
```

Присоединив средство чтения к потоку ввода изображений, можно приступить к вводу из этого потока, вызвав приведенный ниже метод, где `index` — индекс изображений, начиная с нуля.

```
BufferedImage image = reader.read(index);
```

Если поток ввода находится в режиме просмотра данных только в прямом направлении, то изображения должны считываться до тех пор, пока метод `read()` не генерирует исключение типа `IndexOutOfBoundsException`. В противном случае следует вызвать метод `getNumImages()`, как показано ниже.

```
int n = reader.getNumImages(true);
```

Параметр этого метода принимает логическое значение `true`, а это означает, что при вводе разрешается поиск и подсчет количества изображений. Следует, однако, иметь в виду, что метод `getNumImages()` генерирует исключение типа `IllegalStateException`, если поток ввода находится в режиме просмотра данных только в прямом направлении. Данный метод возвращает значение `-1`, если он не в состоянии определить количество изображений без поиска. В этом случае изображения придется считывать до тех пор, пока не возникнет исключение типа `IndexOutOfBoundsException`.

Некоторые файлы могут содержать миниатюрные виды изображений для их предварительного просмотра. Количество миниатюрных видов изображений можно выяснить, сделав следующий вызов:

```
int count = reader.getNumThumbnails(index);
```

А извлечь конкретный миниатюрный вид по его индексу можно следующим образом:

```
BufferedImage thumbnail = reader.getThumbnail(index, thumbnailIndex);
```

Иногда размеры изображения требуется получить еще до его извлечения из файла. Это особенно важно, когда речь идет о крупном изображении или его передаче по низкоскоростному сетевому соединению. Для получения размеров изображения по указанному индексу вызываются следующие методы:

```
int width = reader.getWidth(index);
int height = reader.getHeight(index);
```

Для записи нескольких изображений в файл необходимо создать сначала объект типа `ImageWriter`, а затем с помощью класса `IOImage` перечислить все средства, способные записывать изображения в конкретном формате:

```
String format = . . . ;
ImageWriter writer = null;
Iterator<ImageWriter> iter =
    ImageIO.getImageWritersByFormatName( format );
if (iter.hasNext()) writer = iter.next();
```

Затем поток вывода или файл следует преобразовать в поток вывода изображений (объект типа `ImageOutputStream`) и присоединить его к средству записи следующим образом:

```
File f = . . . ;
ImageOutputStream imageOut = ImageIO.createImageOutputStream(f);
writer.setOutputStream(imageOut);
```

Каждое изображение следует заключить в оболочку объекта типа `IIOImage`, как показано ниже. Дополнительно можно также предоставить список миниатюрных видов и метаданные (например, алгоритм сжатия данных и цветовую информацию). В рассматриваемом здесь примере вместо списка миниатюрных видов и метаданных указываются пустые значения `null`. За дополнительной справкой по данному вопросу обращайтесь к документации на соответствующий прикладной программный интерфейс API.

```
IIOImage iioImage = new IIOImage(images[i], null, null);
```

Для записи первого изображения вызывается метод `write()`:

```
writer.write(new IIOImage(images[0], null, null));
```

А для записи всех последующих изображений служит код из приведенного ниже примера. В качестве третьего параметра метода `write()` может быть указан объект типа `ImageWriteParam`, предоставляющий такие подробности, как мозаичное расположение и алгоритм сжатия данных. В данном примере в качестве третьего параметра указано пустое значение `null`.

```
if (writer.canInsertImage(i))
    writer.writeInsert(i, iioImage, null);
```

Не все форматы допускают сохранение нескольких изображений в файлах. В этом случае метод `canInsertImage()` возвращает логическое значение `false` по условию `i > 0`, и тогда в файле сохраняется только одно изображение. В примере программы из листинга 11.7 изображения загружаются и сохраняются в файлах тех форматов, для которых в библиотеке Java предусмотрены средства чтения и записи. Эта программа позволяет отображать сразу несколько изображений, но не их миниатюрные виды, как показано на рис. 11.27.

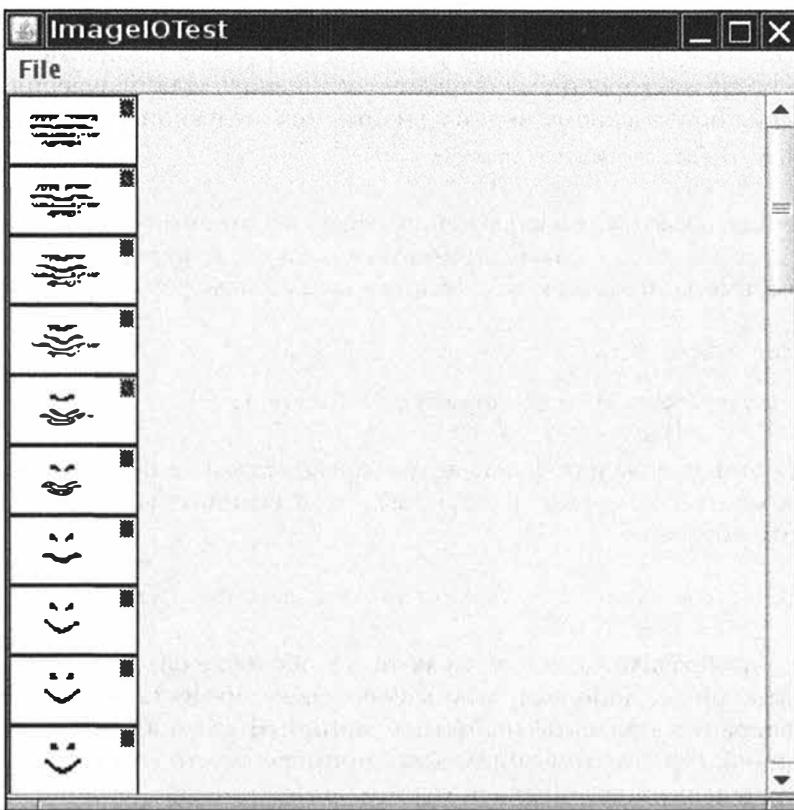


Рис. 11.27. Анимационная последовательность изображений в формате GIF

Листинг 11.7. Исходный код из файла imageIO/ImageIOFrame.java

```
1 package imageIO;
2
3 import java.awt.image.*;
4 import java.io.*;
5 import java.util.*;
6
7 import javax.imageio.*;
8 import javax.imageio.stream.*;
9 import javax.swing.*;
10 import javax.swing.filechooser.*;
11
12 /**
13 * В этом фрейме отображаются загружаемые изображения.
14 * Для загрузки и сохранения изображений в файл
15 * предоставляются отдельные пункты меню
16 */
17 public class ImageIOFrame extends JFrame
18 {
19     private static final int DEFAULT_WIDTH = 400;
20     private static final int DEFAULT_HEIGHT = 400;
21 }
```

```
22 private static Set<String> writerFormats = getWriterFormats();
23
24 private BufferedImage[] images;
25
26 public ImageIOFrame()
27 {
28     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
29
30     JMenu fileMenu = new JMenu("File");
31     JMenuItem openItem = new JMenuItem("Open");
32     openItem.addActionListener(event -> openFile());
33     fileMenu.add(openItem);
34
35     JMenu saveMenu = new JMenu("Save");
36     fileMenu.add(saveMenu);
37     Iterator<String> iter = writerFormats.iterator();
38     while (iter.hasNext())
39     {
40         final String formatName = iter.next();
41         JMenuItem formatItem = new JMenuItem(formatName);
42         saveMenu.add(formatItem);
43         formatItem.addActionListener(event ->
44             saveFile(formatName));
45     }
46
47     JMenuItem exitItem = new JMenuItem("Exit");
48     exitItem.addActionListener(event -> System.exit(0));
49     fileMenu.add(exitItem);
50
51     JMenuBar menuBar = new JMenuBar();
52     menuBar.add(fileMenu);
53     setJMenuBar(menuBar);
54 }
55
56 /**
57 * Открыть файл и загрузить изображения
58 */
59 public void openFile()
60 {
61     JFileChooser chooser = new JFileChooser();
62     chooser.setCurrentDirectory(new File("."));
63     String[] extensions = ImageIO.getReaderFileSuffixes();
64     chooser.setFileFilter(new FileNameExtensionFilter(
65             "Image files", extensions));
66     int r = chooser.showOpenDialog(this);
67     if (r != JFileChooser.APPROVE_OPTION) return;
68     File f = chooser.getSelectedFile();
69     Box box = Box.createVerticalBox();
70     try
71     {
72         String name = f.getName();
73         String suffix = name.substring(name.lastIndexOf('.') + 1);
74         Iterator<ImageReader> iter =
75             ImageIO.getImageReadersBySuffix(suffix);
76         ImageReader reader = iter.next();
77         ImageInputStream imageIn =
78             ImageIO.createImageInputStream(f);
79         reader.setInput(imageIn);
80         int count = reader.getNumImages(true);
81         images = new BufferedImage[count];
```

```
82         for (int i = 0; i < count; i++)
83     {
84         images[i] = reader.read(i);
85         box.add(new JLabel(new ImageIcon(images[i])));
86     }
87 }
88 catch (IOException e)
89 {
90     JOptionPane.showMessageDialog(this, e);
91 }
92 setContentPane(new JScrollPane(box));
93 validate();
94 }
95
96 /**
97 * Сохранить текущее изображение в файле
98 * @param formatName Формат файла
99 */
100 public void saveFile(final String formatName)
101 {
102     if (images == null) return;
103     Iterator<ImageWriter> iter =
104         ImageIO.getImageWritersByFormatName(formatName);
105     ImageWriter writer = iter.next();
106     JFileChooser chooser = new JFileChooser();
107     chooser.setCurrentDirectory(new File("."));
108     String[] extensions =
109         writer.getOriginatingProvider().getFileSuffixes();
110     chooser.setFileFilter(new FileNameExtensionFilter(
111             "Image files", extensions));
112
113     int r = chooser.showSaveDialog(this);
114     if (r != JFileChooser.APPROVE_OPTION) return;
115     File f = chooser.getSelectedFile();
116     try
117     {
118         ImageOutputStream imageOut =
119             ImageIO.createImageOutputStream(f);
120         writer.setOutput(imageOut);
121
122         writer.write(new IIOImage(images[0], null, null));
123         for (int i = 1; i < images.length; i++)
124         {
125             IIOImage iioImage = new IIOImage(images[i], null, null);
126             if (writer.canInsertImage(i))
127                 writer.writeInsert(i, iioImage, null);
128         }
129     }
130     catch (IOException e)
131     {
132         JOptionPane.showMessageDialog(this, e);
133     }
134 }
135
136 /**
137 * Получает ряд предпочтительных названий форматов
138 * из всех средств записи. Предпочтительным считается
139 * первое название формата, указываемое средством записи
140 * @return Возвращает ряд названий форматов
141 */
```

```
142 public static Set<String> getWriterFormats()
143 {
144     Set<String> writerFormats = new TreeSet<>();
145     Set<String> formatNames = new TreeSet<>(
146         Arrays.asList(ImageIO.getWriterFormatNames()));
147     while (formatNames.size() > 0)
148     {
149         String name = formatNames.iterator().next();
150         Iterator<ImageWriter> iter =
151             ImageIO.getImageWritersByFormatName(name);
152         ImageWriter writer = iter.next();
153         String[] names =
154             writer.getOriginatingProvider().getFormatNames();
155         String format = names[0];
156         if (format.equals(format.toLowerCase()))
157             format = format.toUpperCase();
158         writerFormats.add(format);
159         formatNames.removeAll(Arrays.asList(names));
160     }
161     return writerFormats;
162 }
```

javax.imageio.ImageIO 1.4

- **static BufferedImage read(File input)**
- **static BufferedImage read(InputStream input)**
- **static BufferedImage read(URL input)**
Читают изображение из указанного источника ввода данных.
- **static boolean write(RenderedImage image, String formatName, File output)**
- **static boolean write(RenderedImage image, String formatName, OutputStream output)**
Записывают изображение в указанное место назначения вывода данных. Возвращают логическое значение **false**, если соответствующее средство записи не удалось найти.
- **static Iterator<ImageReader> getImageReadersByFormatName(String formatName)**
- **static Iterator<ImageReader> getImageReadersBySuffix(String fileSuffix)**
- **static Iterator<ImageReader> getImageReadersByMIMEType(String mimeType)**
- **static Iterator<ImageWriter> getImageWritersByFormatName(String formatName)**
- **static Iterator<ImageWriter> getImageWritersBySuffix(String fileSuffix)**
- **static Iterator<ImageWriter> getImageWritersByMIMEType(String mimeType)**
Получают все средства чтения или записи, поддерживающие указанный формат (например, JPG), расширение файла (например, jpg) или тип MIME (например, image/jpeg).

javax.imageio.ImageIO 1.4 (окончание)

- **sjavax.imageio.ImageReader 1.4**
- **void setInput(Object input)**
- **void setInput(Object input, boolean seekForwardOnly)**

Устанавливают источник ввода данных для средства чтения.

Параметры: **input**

Объект типа **ImageInputStream** или другой объект, приемлемый для данного средства чтения

seekForwardOnly

Принимает логическое значение **true** для чтения только в прямом направлении. По умолчанию средство чтения использует произвольный доступ и, если требуется, буферизирует данные изображения

- **BufferedImage read(int index)**

Считывает изображение по указанному индексу, начиная с нуля. Если такое изображение отсутствует, генерируется исключение типа **IndexOutOfBoundsException**.

- **int getNumImages(boolean allowSearch)**

Получает количество изображений для данного средства чтения. Если параметр **allowSearch** принимает логическое значение **false** и количество изображений не может быть определено без предварительного просмотра, возвращается значение **-1**. Если же параметр **allowSearch** принимает логическое значение **true** и средство чтения находится в режиме просмотра данных только в прямом направлении, генерируется исключение типа **IllegalStateException**.

- **int getNumThumbnails(int index)**

Получает количество миниатюрных видов изображения по указанному индексу.

- **BufferedImage readThumbnail(int index, int thumbnailIndex)**

Получает по индексу **thumbnailIndex** миниатюрный вид изображения, указанного по индексу **index**.

- **int getWidth(int index)**

- **int getHeight(int index)**

Получают ширину и высоту изображения. Если изображение не найдено, генерируется исключение типа **IndexOutOfBoundsException**.

- **ImageReaderSpi getOriginatingProvider()**

Получает поставщика услуг, создавшего данное средство чтения.

javax.imageio.spi.IIOServiceProvider 1.4

- **String getVendorName()**
- **String getVersion()**

Получают имя и версию данного поставщика услуг.

javax.imageio.spi.ImageReaderWriterSpi 1.4

- `String[] getFormatNames()`
- `String[] getFileSuffixes()`
- `String[] getMIMETypes()`

Получают названия форматов, расширения файлов и типа MIME, которые поддерживаются средствами чтения или записи, созданными данным поставщиком услуг.

javax.imageio.ImageWriter 1.4

- `void setOutput(Object output)`
- Устанавливает место назначения вывода для данного средства записи.

Параметры: `output` Объект типа `ImageOutputStream`
или другой объект, приемлемый
для данного средства записи

- `void write(IIOImage image)`
- `void write(RenderedImage image)`
Записывают одиночное изображение по месту вывода.
- `void writeInsert(int index, IIOImage image, ImageWriteParam param)`
Записывает изображение в файл с несколькими изображениями.
- `boolean canInsertImage(int index)`
Возвращает логическое значение `true`, если в файл можно ввести изображение по указанному индексу.
- `ImageWriterSpi getOriginatingProvider()`
Получает поставщика услуг, создавшего данное средство записи.

javax.imageio.IIOImage 1.4

- `IIOImage(RenderedImage image, List thumbnails, IIOMetadata metadata)`

Создает объект типа `IIOImage`, исходя из указанного изображения, необязательных миниатюрных видов и метаданных.

11.11. Манипулирование изображениями

Допустим, требуется улучшить внешний вид какого-нибудь изображения. Для этого следует получить доступ к отдельным пикселям изображения и заменить их другими. Не исключено также, что придется заново сформировать новые пиксели, например, для отображения результатов физических изменений или математических расчетов. Класс `BufferedImage` дает возможность манипулировать пикселями изображения, а классы, реализующие интерфейс `BufferedImageOp`, — преобразовывать изображения.



На заметку! В версии JDK 1.0 предоставлялась совершенно другая и намного более сложная среда для обработки изображений, которая была оптимизирована для пошагового воспроизведения изображений, построчно загружавшихся из Интернета. Но манипулировать изображениями в такой среде было совсем не просто. Поэтому в данной книге эта среда не рассматривается.

11.11.1. Формирование растровых изображений

Большинство обрабатываемых изображенийчитываются из файла и формируются аппаратными средствами (например, цифровой камерой или сканером) или программными (например, графическими приложениями). В этом разделе рассматривается совершенно другая методика формирования изображений, предполагающая создание в каждый отдельный момент времени только одного пикселя.

Чтобы сформировать изображение, следует сначала создать объект типа `BufferedImage` обычным способом:

```
image = new BufferedImage(width, height,
                           BufferedImage.TYPE_INT_ARGB);
```

Затем необходимо вызвать метод `getRaster()`, как показано ниже, чтобы получить объект типа `WritableRaster`, который служит для доступа к пикселям изображения с целью внести в них изменения.

```
WritableRaster raster = image.getRaster();
```

Метод `setPixel()` позволяет задавать значение цвета отдельного пикселя. Но при его применении возникает следующее осложнение: пикслю нельзя непосредственно присвоить значение цвета типа `Color`. Необходимо знать, каким образом цвета определяются в буферизованном изображении, что зависит от его *типа*. Так, если изображение относится к типу `TYPE_INT_ARGB`, то каждый пиксель описывается четырьмя значениями (в пределах от 0 до 255): для красной, зеленой и синей составляющих, а также для альфа-канала. Эти значения необходимо предоставить в виде массива из четырех элементов целого типа, как показано ниже. В терминах Java 2D API эти значения называются *выборочными значениями* цвета пикселя.

```
int[] black = { 0, 0, 0, 255 };
raster.setPixel(i, j, black);
```



Внимание! Метод `setPixel()` может принимать в качестве параметра массив типа `float[]` или `double[]`. Следует, однако, иметь в виду, что значения цвета, которыми заполняется этот массив, не должны быть нормализованными в пределах от 0 до 1,0, как показано ниже.

```
float[] red = { 1.0F, 0.0F, 0.0F, 1.0F };
raster.setPixel(i, j, red); // ОШИБКА!
```

Независимо от своего типа, массив все равно должен быть заполнен значениями цвета в пределах от 0 до 255.

Для установки значений группы прямоугольных пикселей в качестве параметров метода `setPixels()` можно указать исходное положение, ширину и высоту прямоугольника, а также массив с выборочными значениями цвета этой группы пикселей. Так, если изображение относится к типу `TYPE_INT_ARGB`, сначала следует указать величины красной, зеленой и синей составляющих и значение

прозрачности в альфа-канале для первого пикселя, а затем те же самые данные для второго пикселя и т.д. В приведенном ниже фрагменте кода показано, как это делается.

```
int[] pixels = new int[4 * width * height];
pixels[0] = . . . // значение красной составляющей цвета 1-го пикселя
pixels[1] = . . . // значение зеленої составляющей цвета 1-го пикселя
pixels[2] = . . . // значение синей составляющей цвета 1-го пикселя
pixels[3] = . . . // значение прозрачности в альфа-канале 1-го пикселя
. .
raster.setPixels(x, y, width, height, pixels);
```

А для чтения значения цвета пикселя служит приведенный ниже метод `getPixel()`, возвращающий массив из четырех целых чисел.

```
int[] sample = new int[4];
raster.getPixel(x, y, sample);
Color c = new Color(sample[0], sample[1], sample[2], sample[3]);
```

С помощью метода `getPixels()` можно прочитать значения цвета группы пикселей следующим образом:

```
raster.getPixels(x, y, width, height, samples);
```

Методы `getPixel()` и `setPixel()` можно использовать и для другого типа изображения, если известен способ представления в нем значений цвета пикселей. Иными словами, чтобы применять эти методы, необходимо знать способ кодирования выборочных значений цвета пикселей в конкретном типе изображения.

Манипулировать изображением становится сложнее, если его тип заранее неизвестен. У каждого типа изображений имеется своя *цветовая модель*, в соответствии с которой выборочные значения цвета пикселей можно привести к стандартной цветовой модели RGB.



На заметку! На самом деле цветовая модель RGB не является стандартом. Точное представление цветов зависит от характеристик используемого устройства отображения. Цифровые камеры, сканеры, мониторы и жидкокристаллические дисплеи имеют совершенно разные характеристики. В результате одно и то же значение цвета RGB выглядит по-разному на разных устройствах отображения. Международный консорциум по цвету (International Color Consortium — ICC; <http://www.color.org>) рекомендует сопровождать все цветовые данные цветовым профилем ICC, который определяет степень соответствия цветов стандартной форме их представления по спецификации 1931 CIE XYZ. Эта спецификация разработана Международной комиссией по освещению (Commission Internationale de l'Eclairage — CIE; <http://www.cie.co.at/cie>), обеспечивающей техническую поддержку всех видов освещения и цвета. Она регламентирует стандартный метод представления всех цветов, которые человеческий глаз может воспринять на основе трех гипотетических координат: X, Y, Z. (Подробнее эта спецификация описана в главе 13 упоминавшейся ранее книги *Computer Graphics: Principles and Practice, Second Edition* в С Джеймса Фоли, Андриса ван Дама, Стивена Файнера и др.)

Но цветовые профили ICC имеют очень сложную структуру. Поэтому наряду с ними используется более простой стандарт sRGB (<http://www.w3.org/Graphics/Color/sRGB.html>), который устанавливает точное соответствие значений цветов RGB спецификации 1931 CIE XYZ. Он разработан специально для стандартных цветовых мониторов и используется в прикладном программном интерфейсе Java 2D API для преобразования цветов RGB в значения цветов других цветовых моделей и пространств.

Для получения цветовой модели вызывается метод `getColorModel()`:

```
ColorModel model = image.getColorModel();
```

Для определения цвета пикселя следует вызвать приведенный ниже метод `getDataElements()` из класса `Raster`, который возвращает объект типа `Object` с описанием цвета в соответствии с конкретной цветовой моделью.

```
Object data = raster.getDataElements(x, y, null);
```



На заметку! Объект, который возвращается методом `getDataElements()`, в действительности является массивом выборочных значений цвета пикселя. Это объясняет, почему метод `getDataElements()` называется именно так, а не иначе. Хотя для обработки данного объекта знать об этом совсем не обязательно.

С помощью цветовой модели можно преобразовать полученный объект в стандартные значения цвета ARGB. Метод `getRGB()` возвращает значение типа `int`, состоящее из значений прозрачности в альфа-канале, красной, зеленой и синей составляющих цвета, упакованных в четыре блока по 8 бит в каждом. Из этого целочисленного составного значения цвета с помощью конструктора `Color(int argb, boolean hasAlpha)` можно создать объект типа `Color`:

```
int argb = model.getRGB(data);
Color color = new Color(argb, true);
```

Если же требуется установить определенный цвет пикселя, то описанные выше действия следует выполнить в обратном порядке. Метод `getRGB()` из класса `Color` возвращает целочисленное значение, состоящее из значений прозрачности в альфа-канале, красной, зеленой и синей составляющих цвета. Именно его следует указать в качестве параметра при вызове метода `getDataElements()` из класса `ColorModel`, который, в свою очередь, возвратит объект с описанием цвета пикселя в используемой цветовой модели. Далее этот объект следует передать методу `setDataElements()` из класса `WritableRaster`, как показано ниже.

```
int argb = color.getRGB();
Object data = model.getDataElements(argb, null);
raster.setDataElements(x, y, data);
```

Чтобы проиллюстрировать, каким образом изображение формируется из отдельных пикселей, рассмотрим пример рисования фрактального множества Мандельброта, отдавая дань традиции. Множество Мандельброта образуется путем связывания каждой точки плоскости с некоторой последовательностью цифр. Если последовательность ограничена, точка отмечается цветом. Неокрашенные точки плоскости связаны с неограниченными последовательностями цифр и остаются прозрачными (рис. 11.28).

Ниже показано, как построить простейшее множество Мандельброта. Сначала необходимо отыскать для каждой точки с координатами (a, b) последовательность, которая начинается с выражения $(x, y) = (0, 0)$, и применить к ней следующие формулы:

$$\begin{aligned}x_{\text{new}} &= x^2 - y^2 + a \\y_{\text{new}} &= 2 \cdot x \cdot y + b\end{aligned}$$

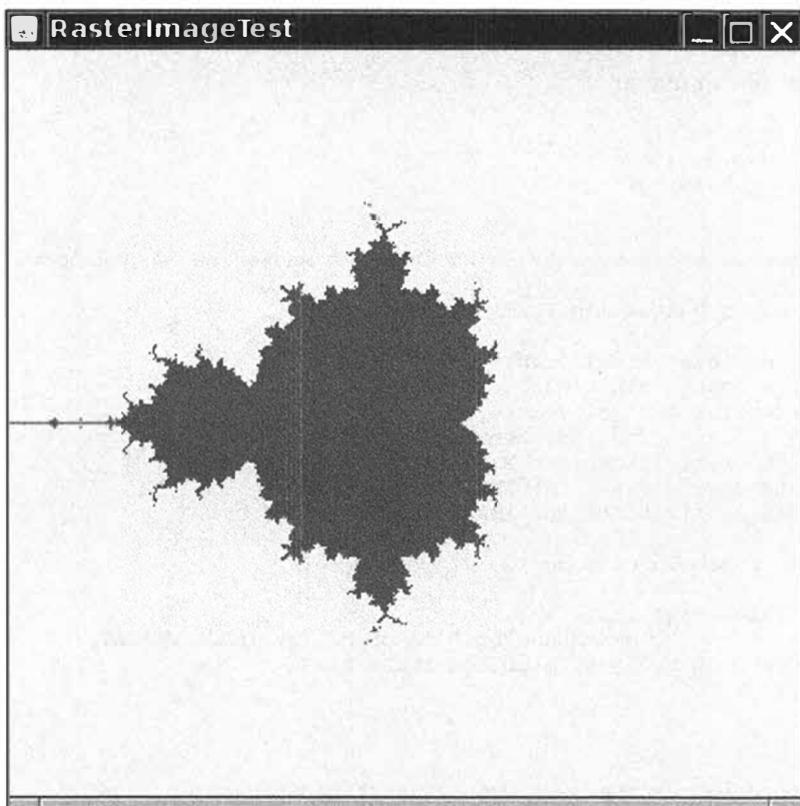


Рис. 11.28. Множество Мандельброта

Если значение x или y оказывается больше 2, последовательность будет превращаться в бесконечную и окрашиваться цветом будут только те пиксели, которые соответствуют точкам с координатами (a, b) , ведущим к ограниченной последовательности. (Формулы для вычисления числовых последовательностей взяты из области математики комплексных чисел в их исходном виде. А дополнительные сведения о математике фракталов можно получить, например, по адресу <http://classes.yale.edu/fractals/>.)

В примере программы, исходный код которой приведен в листинге 11.8, демонстрируется применение класса `ColorModel` для преобразования значений цвета типа `Color` в выборочные данные цвета пикселей. Этот процесс не зависит от типа изображения. Ради интереса попробуйте поэкспериментировать, изменив тип цвета для буферизованного изображения на `TYPE_BYTE_GRAY`. В исходный код данной программы не нужно вносить больше никаких других изменений, поскольку цветовая модель изображения автоматически преобразует цвета пикселей в выборочные значения.

Листинг 11.8. Исходный код из файла rasterImage/RasterImageFrame.java

```
1 package rasterImage;
2
3 import java.awt.*;
4 import java.awt.image.*;
5 import javax.swing.*;
6
7 /**
8  * В этом фрейме показывается изображение множества Мандельброта
9 */
10 public class RasterImageFrame extends JFrame
11 {
12     private static final double XMIN = -2;
13     private static final double XMAX = 2;
14     private static final double YMIN = -2;
15     private static final double YMAX = 2;
16     private static final int MAX_ITERATIONS = 16;
17     private static final int IMAGE_WIDTH = 400;
18     private static final int IMAGE_HEIGHT = 400;
19
20     public RasterImageFrame()
21     {
22         BufferedImage image =
23             makeMandelbrot(IMAGE_WIDTH, IMAGE_HEIGHT);
24         add(new JLabel(new ImageIcon(image)));
25         pack();
26     }
27
28 /**
29  * Формирует изображение множества Мандельброта
30  * @param width Ширина изображения
31  * @param height Высота изображения
32  * @return Возвращает изображение
33 */
34 public BufferedImage makeMandelbrot(int width, int height)
35 {
36     BufferedImage image = new BufferedImage(width, height,
37             BufferedImage.TYPE_INT_ARGB);
38     WritableRaster raster = image.getRaster();
39     ColorModel model = image.getColorModel();
40
41     Color fractalColor = Color.red;
42     int argb = fractalColor.getRGB();
43     Object colorData = model.getDataElements(argb, null);
44
45     for (int i = 0; i < width; i++)
46         for (int j = 0; j < height; j++)
47     {
48         double a = XMIN + i * (XMAX - XMIN) / width;
49         double b = YMIN + j * (YMAX - YMIN) / height;
50         if (!escapesToInfinity(a, b))
51             raster.setDataElements(i, j, colorData);
52     }
53     return image;
54 }
55
56 private boolean escapesToInfinity(double a, double b)
```

```

57  {
58      double x = 0.0;
59      double y = 0.0;
60      int iterations = 0;
61      while (x <= 2 && y <= 2 && iterations < MAX_ITERATIONS)
62      {
63          double xnew = x * x - y * y + a;
64          double ynew = 2 * x * y + b;
65          x = xnew;
66          y = ynew;
67          iterations++;
68      }
69      return x > 2 || y > 2;
70  }
71 }
```

java.awt.image.BufferedImage 1.2

- **BufferedImage(int width, int height, int imageType)**

Создает объект буферизованного изображения.

Параметры:

width, height Размеры изображения

imageType

Тип изображения. К числу наиболее распространенных относятся следующие типы изображений:

TYPE_INT_RGB,
TYPE_INT_ARGB,
TYPE_BYTE_GRAY и
TYPE_BYTE_INDEXED

- **ColorModel getColorModel()**

Возвращает цветовую модель данного буферизованного изображения.

- **WritableRaster getRaster()**

Возвращает растр для доступа к пикселям данного буферизованного изображения с целью их изменения.

java.awt.image.Raster 1.2

- **Object getDataElements(int x, int y, Object data)**

Возвращает для точки раstra выборочные данные в виде массива, тип элементов и длина которого зависят от цветовой модели. Если параметр **data** не принимает пустое значение **null**, то предполагается, что это значение является массивом для хранения выборочных данных, причем массив заполнен. А если параметр **data** принимает пустое значение **null**, то создается новый массив. Тип его элементов и длина зависят от цветовой модели.

- **int[] getPixel(int x, int y, int[] sampleValues)**

- **float[] getPixel(int x, int y, float[] sampleValues)**

- **double[] getPixel(int x, int y, double[] sampleValues)**

java.awt.image.Raster 1.2 (окончание)

- `int[] getPixels(int x, int y, int width, int height, int[] sampleValues)`
- `float[] getPixels(int x, int y, int width, int height, float[] sampleValues)`
- `double[] getPixels(int x, int y, int width, int height, double[] sampleValues)`

Возвращают выборочные значения для точки растра или группы точек, составляющих прямоугольную область. Эти значения помещаются в массив, длина которого зависит от конкретной цветовой модели. Если параметр `sampleValues` принимает пустое значение `null`, то создается новый массив. Применяются только в том случае, если заранее известно, каким образом выборочные значения определяются для цветовой модели.

java.awt.image.WritableRaster 1.2

- `void setDataElements(int x, int y, Object data)`

Задает выборочные данные для точки растра. Параметр `data` обозначает массив выборочных значений цвета пикселя. Тип элементов и длина этого массива зависят от конкретной цветовой модели.

- `void setPixel(int x, int y, int[] sampleValues)`
- `void setPixel(int x, int y, float[] sampleValues)`
- `void setPixel(int x, int y, double[] sampleValues)`
- `void setPixels(int x, int y, int width, int height, int[] sampleValues)`
- `void setPixels(int x, int y, int width, int height, float[] sampleValues)`
- `void setPixels(int x, int y, int width, int height, double[] sampleValues)`

Устанавливают выборочные значения для точки растра или группы точек, составляющих прямоугольную область. Применяются только в том случае, если заранее известен порядок кодировки выборочных значений для цветовой модели.

java.awt.image.ColorModel 1.2

- `int getRGB(Object data)`

Возвращает значение цвета ARGB, которое соответствует выборочным данным, переданным в массиве `data`, тип элементов и длина которого зависят от конкретной цветовой модели.

- `Object getDataElements(int argb, Object data)`

Возвращает выборочные данные для указанного значения цвета. Если параметр `data` не принимает пустое значение `null`, то предполагается, что это массив, имеющий подходящую длину для хранения данных и заполненный ими. А если параметр `data` принимает пустое значение `null`, то создается новый массив, который заполняется выборочными данными цвета пикселя. Тип его элементов и длина массива зависят от цветовой модели.

```
java.awt.Color 1.0
```

- **Color(int argb, boolean hasAlpha)** 1.2

Формирует цвет по указанному составному значению ARGB, если параметр **hasAlpha** принимает логическое значение **true**, или же по указанному стандартному значению RGB, если параметр **hasAlpha** принимает логическое значение **false**.

- **int getRGB()**

Возвращает значение ARGB, соответствующее данному цвету.

11.11.2. Фильтрация изображений

В предыдущем разделе рассматривался процесс формирования изображения сызнова. Но нередко требуется обработать уже имеющееся изображение. Безусловно, данные изображения можно сначала прочитать, используя методы `getPixel()/getDataElements()`, представленные в предыдущем разделе, затем преобразовать их и записать обратно. Но, к счастью, в прикладном программном интерфейсе Java 2D API поддерживается целый ряд фильтров, автоматически выполняющих многие рутинные операции обработки изображений.

Все классы, выполняющие операции манипулирования изображениями, реализуют интерфейс `BufferedImageOp`. Для преобразования одного изображения в другое после создания объекта требующейся операции достаточно вызвать метод `filter()` следующим образом:

```
BufferedImageOp op = . . .;
BufferedImage filteredImage = new BufferedImage
    (image.getWidth(), image.getHeight(), image.getType());
op.filter(image, filteredImage);
```

Некоторые методы, например `op.filter(image, image)`, могут непосредственно выполнять преобразования изображений, но большинство остальных методов не способны на это. Ниже перечислены пять классов, реализующих интерфейс `BufferedImageOp`.

```
AffineTransformOp
RescaleOp
LookupOp
ColorConvertOp
ConvolveOp
```

В частности, класс `AffineTransformOp` выполняет аффинное преобразование пикселей. Например, в приведенном ниже фрагменте кода показано, как повернуть изображение вокруг его центра.

```
AffineTransform transform = AffineTransform.getRotateInstance(
    Math.toRadians(angle), image.getWidth() / 2,
    image.getHeight() / 2);
AffineTransformOp op = new AffineTransformOp(
    transform, interpolation);
op.filter(image, filteredImage);
```

При вызове конструктора класса `AffineTransformOp` в качестве параметров следует указать аффинное преобразование и алгоритм интерполяции. Интерполяция требуется для определения местоположения пикселей в целевом

изображении, если исходные пиксели оказываются после преобразования где-то между целевыми пикселями.

Например, после вращения исходные пиксели точно совпадают с целевыми пикселями. Для интерполяции используются два алгоритма: билинейный (`AffineTransformOp.TYPE_BILINEAR`) и ближайшего соседа (`AffineTransformOp.TYPE_NEAREST_NEIGHBOR`). Билинейная интерполяция выполняется дольше, но позволяет добиться более качественного внешнего вида преобразованного изображения.

В примере программы из листинга 11.9 изображение поворачивается на 5° (рис. 11.29). В частности, класс `RescaleOp` выполняет для каждой составляющей цвета изображения следующую операцию по изменению масштаба:

$$x_{\text{new}} = a \cdot x + b$$

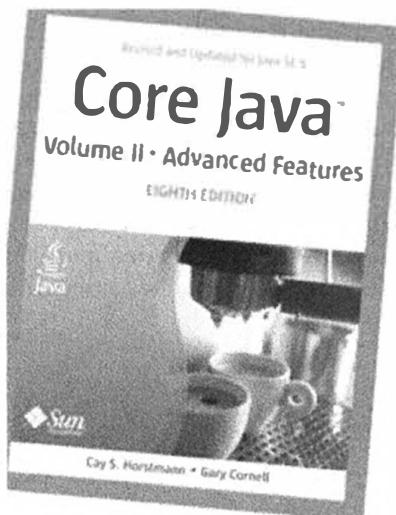


Рис. 11.29. Вращение изображения

(На составляющие прозрачности из альфа-канала эта операция не распространяется). В результате изменения масштаба на величину $a > 1$ увеличивается яркость изображения. Объект типа `RescaleOp` создается по указанным параметрам масштабирования и дополнительным, но не обязательным параметрам рисования. В примере программы из листинга 11.9 для этой цели используется приведенный ниже фрагмент кода. Кроме того, отдельные величины изменения масштаба можно предоставить для каждой составляющей цвета, как поясняется далее при описании прикладного программного интерфейса API.

```
float a = 1.1f;
float b = 20.0f;
RescaleOp op = new RescaleOp(a, b, null);
```

Конструктор класса `LookupOp` позволяет указать произвольное отображение выборочных значений цвета. Для этого достаточно предоставить таблицу, в которой указаны правила отображения каждого выборочного значения цвета. В рассматриваемом здесь примере формируется *негатив* исходного изображения. Для этого каждый цвет с заменяется цветом 255 – с.

В качестве параметров конструктору класса `LookupOp` требуется указать объект типа `LookupTable` и таблицу с необязательными указаниями по воспроизведению. Класс `LookupTable` является абстрактным и имеет два конкретных подкласса: `ByteLookupTable` и `ShortLookupTable`. На первый взгляд, для целей преобразования было бы достаточно и класса `ByteLookupTable`, поскольку значения цвета RGB представлены в виде байтов. Но из-за программной ошибки, подробно описанной в документации по адресу http://bugs.java.com/bugdatabase/view_bug.do?bug_id=6183251, вместо него в данном примере используется класс `ShortLookupTable`. В приведенном ниже фрагменте кода показано, каким образом формируется операция табличного поиска для преобразования цветного изображения в негатив.

```
short negative[] = new short[256];
for (int i = 0; i < 256; i++) negative[i] = (short) (255 - i);
ShortLookupTable table = new ShortLookupTable(0, negative);
LookupOp op = new LookupOp(table, null);
```

Данная операция выполняется над каждой составляющей цвета в отдельности, но не над составляющими прозрачности в альфа-канале. Для преобразования каждой составляющей цвета можно предоставить и другие таблицы поиска (подробнее об этом см. далее в описании прикладного программного интерфейса API).



На заметку! Класс `LookupOp` нельзя применять к изображениям с индексированной цифровой моделью, потому что в таких изображениях каждое выборочное значение цвета пикселей задается в виде смещения в палитре цветов.

Для преобразования цветового пространства служит класс `ColorConvertOp`, который здесь не рассматривается. А для наиболее глубоких преобразований предназначен класс `ConvolveOp`, который выполняет математическую операцию *свертки*. Мы не будем углубляться в математические подробности выполнения операции свертки, но понять основную идею несложно. В качестве примера рассмотрим фильтр размытия, действие которого показано на рис. 11.30.

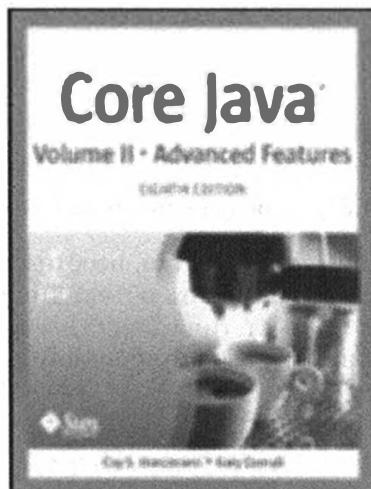


Рис. 11.30. Размытие изображения

Размытие достигается путем замены каждого пикселя средним значением этого пикселя и соседних с ним восьми пикселей. Интуитивно понятно, почему эта операция делает изображение размытым, а с математической точки зрения получение среднего значения может быть представлено как операция свертки со следующим ядром:

$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

Ядро свертки представляет собой матрицу с весами соседних значений. Например, приведенное выше ядро приводит к размытию изображения. А представленное ниже ядро выполняет операцию *определения краев*, т.е. определяет участки, на которых происходит изменение цвета. Алгоритм определения краев играет важную роль в обработке фотоизображений (рис. 11.31).

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$



Рис. 11.31. Определение краев и инверсия

Чтобы сформировать операцию свертки, необходимо сначала составить массив значений для ядра свертки и создать объект типа *Kernel*. Затем на основе этого ядра следует создать объект типа *ConvolveOp*, чтобы использовать его для фильтрации изображения следующим образом:

```
float[] elements =
{
    0.0f, -1.0f, 0.0f,
    -1.0f, 4.f, -1.0f,
    0.0f, -1.0f, 0.0f
};
Kernel kernel = new Kernel(3, 3, elements);
```

```
ConvolveOp op = new ConvolveOp(kernel);
op.filter(image, filteredImage);
```

В примере программы из листинга 11.9 пользователю предоставляется возможность загружать файлы формата GIF и JPEG и выполнять над ними описанные выше операции. Простота кода этой программы объясняется тем, что в прикладном программном интерфейсе Java 2D API предусмотрены весьма эффективные средства для обработки изображений.

Листинг 11.9. Исходный код из файла ImageProcessing/ImageProcessingFrame.java

```
1 package imageProcessing;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import java.awt.image.*;
6 import java.io.*;
7
8 import javax.imageio.*;
9 import javax.swing.*;
10 import javax.swing.filechooser.*;
11
12 /**
13 * Этот фрейм содержит меню для загрузки изображения и
14 * выбора различных его преобразований, а также компонент
15 * для показа итогового изображения
16 */
17 public class ImageProcessingFrame extends JFrame
18 {
19     private static final int DEFAULT_WIDTH = 400;
20     private static final int DEFAULT_HEIGHT = 400;
21
22     private BufferedImage image;
23
24     public ImageProcessingFrame()
25     {
26         setTitle("ImageProcessingTest");
27         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
28
29         add(new JComponent()
30         {
31             public void paintComponent(Graphics g)
32             {
33                 if (image != null) g.drawImage(image, 0, 0, null);
34             }
35         });
36
37         JMenu fileMenu = new JMenu("File");
38         JMenuItem openItem = new JMenuItem("Open");
39         openItem.addActionListener(event -> openFileDialog());
40         fileMenu.add(openItem);
41
42         JMenuItem exitItem = new JMenuItem("Exit");
43         exitItem.addActionListener(event -> System.exit(0));
44         fileMenu.add(exitItem);
45
46         JMenu editMenu = new JMenu("Edit");
47         JMenuItem blurItem = new JMenuItem("Blur");
48         blurItem.addActionListener(event ->
49             {
```

```
50         float weight = 1.0f / 9.0f;
51         float[] elements = new float[9];
52         for (int i = 0; i < 9; i++)
53             elements[i] = weight;
54         convolve(elements);
55     });
56     editMenu.add(blurItem);
57
58     JMenuItem sharpenItem = new JMenuItem("Sharpen");
59     sharpenItem.addActionListener(event ->
60     {
61         float[] elements = { 0.0f, -1.0f, 0.0f, -1.0f,
62                             5.0f, -1.0f, 0.0f, -1.0f, 0.0f };
63         convolve(elements);
64     });
65     editMenu.add(sharpenItem);
66
67     JMenuItem brightenItem = new JMenuItem("Brighten");
68     brightenItem.addActionListener(event ->
69     {
70         float a = 1.1f;
71         float b = 20.0f;
72         RescaleOp op = new RescaleOp(a, b, null);
73         filter(op);
74     });
75     editMenu.add(brightenItem);
76
77     JMenuItem edgeDetectItem = new JMenuItem("Edge detect");
78     edgeDetectItem.addActionListener(event ->
79     {
80         float[] elements = { 0.0f, -1.0f, 0.0f, -1.0f, 4.0f,
81                             -1.0f, 0.0f, -1.0f, 0.0f };
82         convolve(elements);
83     });
84     editMenu.add(edgeDetectItem);
85
86     JMenuItem negativeItem = new JMenuItem("Negative");
87     negativeItem.addActionListener(event ->
88     {
89         short[] negative = new short[256 * 1];
90         for (int i = 0; i < 256; i++)
91             negative[i] = (short) (255 - i);
92         ShortLookupTable table =
93             new ShortLookupTable(0, negative);
94         LookupOp op = new LookupOp(table, null);
95         filter(op);
96     });
97     editMenu.add(negativeItem);
98
99     JMenuItem rotateItem = new JMenuItem("Rotate");
100    rotateItem.addActionListener(event ->
101    {
102        if (image == null) return;
103        AffineTransform transform = AffineTransform
104            .getRotateInstance(Math.toRadians(5),
105                            image.getWidth() / 2,
106                            image.getHeight() / 2);
107        AffineTransformOp op = new AffineTransformOp(
108            transform, AffineTransformOp.TYPE_BICUBIC);
109        filter(op);
```

```
110     });
111     editMenu.add(rotateItem);
112
113     JMenuBar menuBar = new JMenuBar();
114     menuBar.add(fileMenu);
115     menuBar.add(editMenu);
116     setJMenuBar(menuBar);
117 }
118
119 /**
120 * Открыть файл и загрузить изображение
121 */
122 public void openFile()
123 {
124     JFileChooser chooser = new JFileChooser(".");
125     chooser.setCurrentDirectory(new File(getClass()
126             .getPackage().getName()));
127     String[] extensions = ImageIO.getReaderFileSuffixes();
128     chooser.setFileFilter(new FileNameExtensionFilter(
129             "Image files", extensions));
130     int r = chooser.showOpenDialog(this);
131     if (r != JFileChooser.APPROVE_OPTION) return;
132
133     try
134     {
135         Image img = ImageIO.read(chooser.getSelectedFile());
136         image = new BufferedImage(img.getWidth(null),
137             img.getHeight(null),
138             BufferedImage.TYPE_INT_RGB);
139         image.getGraphics().drawImage(img, 0, 0, null);
140     }
141     catch (IOException e)
142     {
143         JOptionPane.showMessageDialog(this, e);
144     }
145     repaint();
146 }
147
148 /**
149 * Применить фильтр и перерисовать
150 * @param op Выполняемая операция преобразования
151 */
152 private void filter(BufferedImageOp op)
153 {
154     if (image == null) return;
155     image = op.filter(image, null);
156     repaint();
157 }
158
159 /**
160 * Выполнить свертку и перерисовать
161 * @param elements Ядро свертки (массив из 9 элементов матрицы)
162 */
163 private void convolve(float[] elements)
164 {
165     Kernel kernel = new Kernel(3, 3, elements);
166     ConvolveOp op = new ConvolveOp(kernel);
167     filter(op);
168 }
169 }
```

java.awt.image.BufferedImageOp 1.2

- **BufferedImage filter(BufferedImage source, BufferedImage dest)**

Выполняет операцию над исходным изображением *source* и сохраняет (а также возвращает) результат в виде изображения *dest*. Если в качестве параметра *dest* указано пустое значение *null*, то создается новое целевое изображение, которое затем возвращается.

java.awt.image.AffineTransformOp 1.2

- **AffineTransformOp(AffineTransform t, int interpolationType)**

Создает объект для операции аффинного преобразования. В качестве алгоритма интерполяции может быть указано значение одной из следующих констант: **TYPE_BILINEAR**, **TYPE_BICUBIC** или **TYPE_NEAREST_NEIGHBOR**.

java.awt.image.RescaleOp 1.2

- **RescaleOp(float a, float b, RenderingHints hints)**
- **RescaleOp(float[] as, float[] bs, RenderingHints hints)**

Создают объект для следующей операции изменения масштаба: $x_{\text{new}} = a \cdot x + b$. При использовании первого конструктора все составляющие цвета (кроме составляющей прозрачности из альфа-канала) масштабируются с одинаковыми коэффициентами. А при использовании второго конструктора значения предоставляются для каждой составляющей цвета в отдельности, но не затрагивая составляющую прозрачности из альфа-канала, или же для составляющих как цвета, так и прозрачности из альфа-канала.

java.awt.image.LookupOp 1.2

- **LookupOp(LookupTable table, RenderingHints hints)**

Создает объект операции поиска для указанной таблицы поиска.

java.awt.image.ByteLookupTable 1.2

- **ByteLookupTable(int offset, byte[] data)**
- **ByteLookupTable(int offset, byte[][] data)**

Создают таблицу поиска для преобразования значений типа **byte**. Значение смещения вычитается из входных данных до преобразования. Значения в первом конструкторе применяются ко всем составляющим цвета, но не к составляющей прозрачности из альфа-канала. При использовании второго конструктора значения предоставляются для каждой составляющей цвета в отдельности, но не затрагивая составляющую прозрачности из альфа-канала, или же для составляющих как цвета, так и прозрачности из альфа-канала.

java.awt.image.ShortLookupTable 1.2

- `ShortLookupTable(int offset, short[] data)`
- `ShortLookupTable(int offset, short[][] data)`
- Создают таблицу поиска для преобразования значений типа `short`. Значение смещения вычитается из входных данных до преобразования. Значения в первом конструкторе применяются ко всем составляющим цвета, но не к составляющей прозрачности из альфа-канала. При использовании второго конструктора значения предоставляются для каждой составляющей цвета в отдельности, но не затрагивая составляющую прозрачности из альфа-канала, или же для составляющих как цвета, так и прозрачности из альфа-канала.

java.awt.image.ConvolveOp 1.2

- `ConvolveOp(Kernel kernel)`
- `ConvolveOp(Kernel kernel, int edgeCondition, RenderingHints hints)`
Создают объект для операции свертки. В качестве параметра `edgeCondition` может быть указано значение одной из следующих констант: `EDGE_NO_OP` или `EDGE_ZERO_FILL`. С этими значениями следует быть особенно внимательным, поскольку у них нет достаточного количества соседних значений для вычисления свертки. По умолчанию устанавливается значение константы `EDGE_ZERO_FILL`.

java.awt.image.Kernel 1.2

- `Kernel(int width, int height, float[] matrixElements)`
Создает ядро свертки для указанной матрицы.

11.12. Вывод изображений на печать

В исходной версии JDK вообще не поддерживался вывод на печать. Например, нельзя было печатать из аплетов, а для печати из приложений необходимо было пользоваться дополнительными библиотеками независимых поставщиков. Впервые средства для организации вывода на печать появились в версии JDK 1.1, но с их помощью можно было выводить очень простые документы довольно низкого качества. Разработчикам веб-браузеров была предоставлена специальная модель печати, позволявшая выводить окно аплета в таком виде, в каком оно отображалось на веб-странице (чем они так и не воспользовались).

В версии Java SE 1.2 были заложены основы надежной модели печати, полностью интегрированной со средствами двухмерной графики. А в версии Java SE 1.4 эта модель была существенно усовершенствована и дополнена возможностями управления параметрами печатающих устройств и организации заданий потоковой печати на стороне сервера.

В этом разделе показано, каким образом изображения печатаются на одной странице бумаги, как организовать многостраничную печать, а также демонстрируются возможности модели воспроизведения изображений в прикладном

программном интерфейсе Java 2D API и средства для создания окна предварительного просмотра печати.

11.12.1. Вывод двухмерной графики на печать

В данном разделе рассматривается одна из наиболее распространенных задач — печать двухмерной графики. Разумеется, изображение может содержать фрагменты текста, отформатированного различными шрифтами, или даже полностью состоять из текста. Для вывода двухмерной графики на печать необходимо выполнить следующие действия:

- Предоставить объект, класс которого реализует интерфейс `Printable`.
- Запустить задание на печать.

В интерфейсе `Printable` объявлен единственный метод `print()`:

```
int print(Graphics g, PageFormat format, int page)
```

который вызывается всякий раз, когда механизм печати форматирует страницу. Прикладной код рисует текст и изображение, которые должны быть напечатаны в заданном графическом контексте `g`. Формат страницы (параметр `format`) определяет формат бумаги и поля для печати, а номер страницы (параметр `page`) служит для выбора воспроизводимой страницы.

Для запуска задания на печать служит класс `PrinterJob`. Сначала для получения объекта задания на печать вызывается статический метод `getPrinterJob()`, а затем с помощью метода `setPrintable()` указывается печатаемый объект типа `Printable`:

```
Printable canvas = . . .;
PrinterJob job = PrinterJob.getPrinterJob();
job.setPrintable(canvas);
```

 **Внимание!** Не путайте класс `PrinterJob` с устаревшим классом `PrintJob`, который управлял процессом печати в версии JDK 1.1.

Прежде чем запускать задание на печать, следует вызвать метод `printDialog()`, чтобы открыть диалоговое окно, приведенное на рис. 11.32. В этом окне пользователю предоставляется возможность выбрать устройство для вывода на печать (если доступно несколько печатающих устройств), указать диапазон печатаемых страниц и сделать прочие настройки печати.

Все параметры печатающего устройства обычно собираются в экземпляре класса, реализующего интерфейс `PrintRequestAttributeSet`, например класса `HashPrintRequestAttributeSet`:

```
HashPrintRequestAttributeSet attributes =
    new HashPrintRequestAttributeSet();
```

Атрибуты печати должны быть переданы методу `printDialog()` в виде объекта `attributes`. Метод `printDialog()` возвращает логическое значение `true`, если пользователь подтверждает выбор параметров печати, или логическое значение `false`, если пользователь отменяет внесенные изменения. В первом случае вызывается метод `print()` из класса `PrinterJob` для запуска задания на печать. Он

может генерировать исключение типа `PrinterException`. В приведенном ниже фрагменте кода показано, каким образом вывод на печать организуется в общих чертах.

```
if (job.printDialog(attributes))
{
    try
    {
        job.print(attributes);
    }
    catch (PrinterException exception)
    {
        ...
    }
}
```

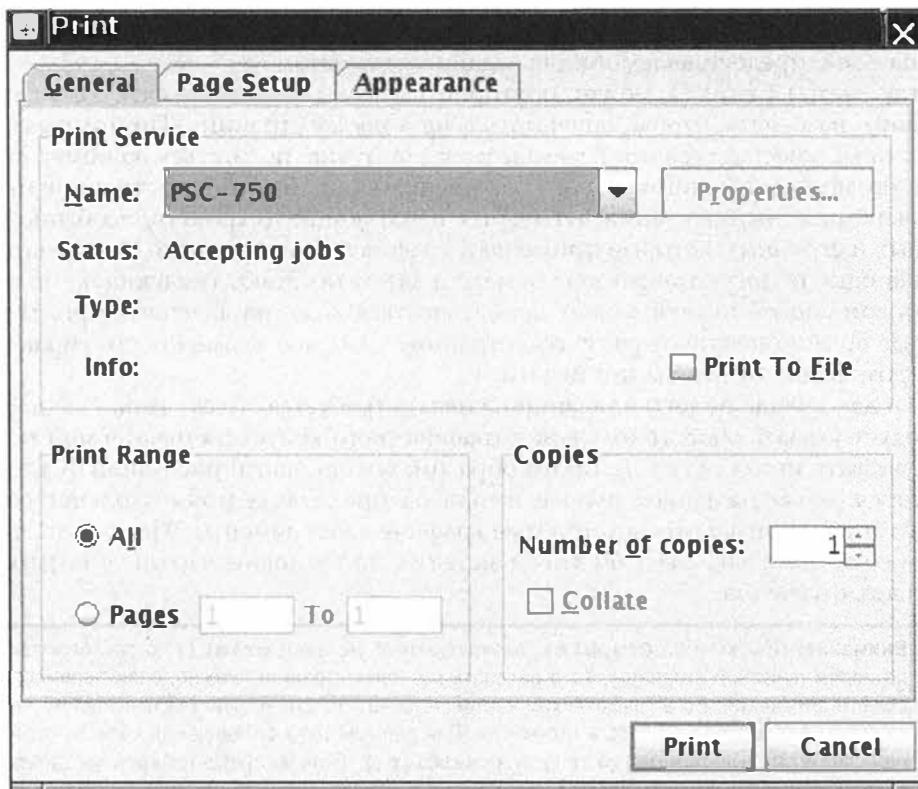


Рис. 11.32. Диалоговое окно для настройки печати независимо от используемой платформы



На заметку! До выпуска версии JDK 1.4 в механизме печати Java использовались диалоговые окна настройки печати, ориентированные именно на ту платформу, на которой работал пользователь. Для отображения платформенно-ориентированного диалогового окна настройки печати следует вызвать метод `printDialog()` без указания параметров. [В этом случае пользовательские установки нельзя объединить в набор атрибутов печати.]

Во время печати метод `print()` из класса `PrinterJob` неоднократно вызывает метод `print()` для объекта типа `Printable`, связанного с текущим заданием на печать. А поскольку в задании на печать неизвестно количество выводимых страниц, то метод `print()` вызывается повторно, если он возвращает значение константы `Printable.PAGE_EXISTS`. Печать завершится лишь тогда, когда метод `print()` возвратит значение константы `Printable.NO_SUCH_PAGE`.

 **Внимание!** Отсчет номеров страниц, передаваемых методу `print()`, начинается с нуля.

Таким образом, задание на печать не может определить количество страниц до тех пор, пока печать не будет завершена. Поэтому в диалоговом окне настройки печати не может быть представлен фактический диапазон номеров страниц, а вместо него выводится сообщение "Pages 1 to 1" (Страниц от 1 до 1). В следующем разделе будет показано, как этот недостаток устраняется с помощью объекта типа `Book`, предоставляемого для задания на печать.

Итак, метод `print()` может повторно вызываться для объекта `Printable` в задании на печать, чтобы напечатать *одну* и *ту же* страницу. Поэтому вместо подсчета количества страниц в методе `print()` лучше полагаться на номер страницы. Возможность многократного обращения к одной и той же странице предназначена для обслуживания некоторых печатающих устройств, особенно матричных и струйных, которые применяют *полосовой* способ печати. Они печатают сначала одну полосу, продвигают бумагу и затем печатают следующую полосу. Полосовой способ печати может использоваться даже на лазерных принтерах, которые обычно печатают сразу всю страницу. Это дает возможность управлять размером файла буферизации печати.

Если для вывода полосы в задании на печать требуется объект типа `Printable`, то следует указать область отсечения графического контекста печатаемой полосой и вызвать метод `print()`. Таким образом, все операции рисования будут выполняться только на данной полосе, и только в пределах ее прямоугольной области будут воспроизводиться рисуемые графические элементы. Методу `print()` совсем не обязательно знать об этих действиях, при условии, что он не затрагивает область отсечения.

 **Внимание!** Объект типа `Graphics`, возвращаемый методом `print()`, отсекается также по полям печатной страницы. Хотя рисовать на полях страницы можно, если переместить область отсечения. Но в графическом контексте печатающего устройства необходимо строго соблюдать заданную область отсечения. Для дальнейшего ограничения области отсечения следует вызывать метод `clip()`, а не `setClip()`. Если же требуется временно удалить область отсечения, то в начале разрабатываемого метода `print()` придется вызвать метод `getClip()`, чтобы получить эту область для ее последующего восстановления.

Параметр типа `PageFormat` метода `print()` содержит сведения о печатаемой странице. Методы `getWidth()` и `getHeight()` возвращают формат бумаги, измеряемый в *пунктах*. Один пункт равен 1/72 дюйма, а один дюйм составляет 25,4 миллиметра. Например, формат бумаги A4 приблизительно равен 595×842 пункта, или 210×297 мм, а формат бумаги US Letter 612×792 пункта, или 215,9×279,4 мм.

Пункты являются общепринятой единицей измерения в полиграфической промышленности США. Эта же единица применяется по умолчанию для всех

графических контекстов печати. В этом можно убедиться на примере программы, исходный код которой приведен в конце данного раздела. Эта программа печатает две строки текста, которые находятся на расстоянии 72 пунктов друг от друга. Запустите эту программу, измерьте расстояние между напечатанными строками и убедитесь в том, что оно равно точно 1 дюйму, или 25,4 мм.

Методы `getWidth()` и `getHeight()` из класса `PageFormat` возвращают полную ширину и высоту страницы. Но печатать можно не на всей странице. Обычно пользователи самостоятельно задают размеры полей печатаемой страницы, но даже если они этого не сделают, то все равно поля будут определены. Дело в том, что печатающему устройству нужно каким-то образом удерживать листы бумаги, на которых производится печать, и поэтому по краям остаются небольшие непечатаемые участки.

Методы `getImageableWidth()` и `getImageableHeight()` возвращают ширину и высоту области, доступной для печати. Но поля не обязательно должны быть симметричными, поэтому нужно также знать координаты верхнего левого угла области, доступной для печати, которая условно показана на рис. 11.33. Эти координаты можно получить с помощью методов `getImageableX()` и `getImageableY()`.



Рис. 11.33. Размеры страницы и области, доступной для печати



Совет! Графический контекст, который передается методу `print()`, обрезается по полям печатаемой страницы, тем не менее начало системы координат остается по-прежнему в верхнем левом углу страницы. Поэтому начало координат имеет смысл перенести в левый верхний угол области, доступной для печати. Для этого разрабатываемый метод `print()` нужно начать со следующей строки кода:

```
g.translate(pageFormat.getImageableX(), pageFormat.getImageableY());
```

Если же требуется предоставить пользователю возможность определять границы страницы и задавать книжную или альбомную ориентацию, не устанавливая других атрибутов печати, то для этого можно вызвать метод `pageDialog()` из класса `PrinterJob` следующим образом:

```
PageFormat format = job.pageDialog(attributes);
```



На заметку! На одной из вкладок диалогового окна настройки печати содержатся параметры печатаемой страницы (рис. 11.34), которые пользователь может просмотреть и поправить непосредственно перед печатью. Это особенно полезно, если в программе реализован принцип WYSIWIG, когда на отпечатке получается именно то, что отображается на экране. Метод `pageDialog()` возвращает объект типа `PageFormat` со значениями параметров печати, задаваемых пользователем.

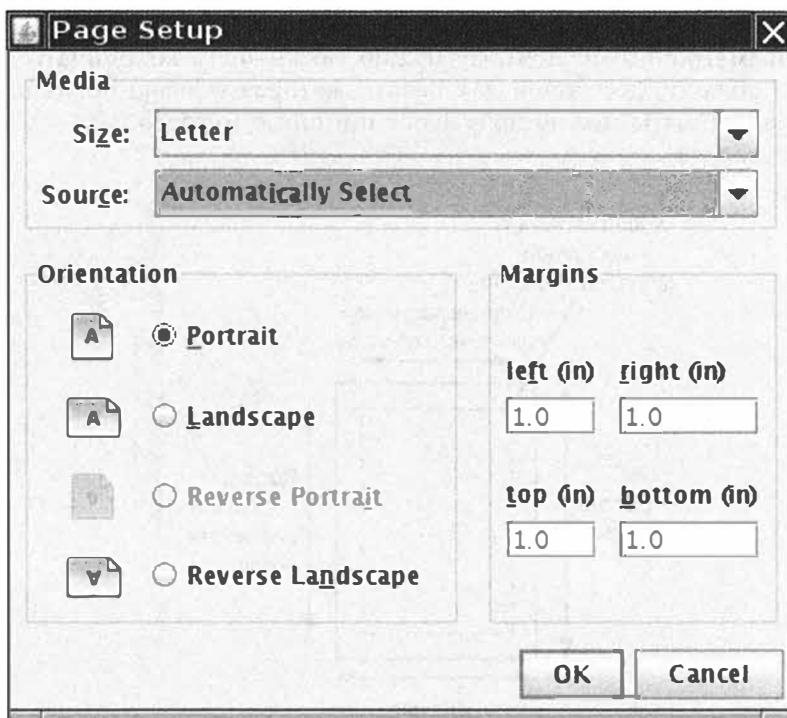


Рис. 11.34. Диалоговое окно для настройки печатаемой страницы независимо от используемой платформы

В примере программы, исходный код которой приведен в листингах 11.10 и 11.11, показано, каким образом один и тот же ряд фигур выводится на экран и на печать. Класс `PrintPanel` расширяет класс `JPanel` и реализует интерфейс `Printable`, а его методы `paintComponent()` и `print()` вызывают один и тот же метод непосредственно для рисования изображения, как выделено ниже полужирным.

```
class PrintPanel extends JPanel implements Printable
{
    public void paintComponent(Graphics g)
```

```

    {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D) g;
        drawPage(g2);
    }

    public int print(Graphics g, PageFormat pf, int page)
        throws PrinterException
    {
        if (page >= 1) return Printable.NO_SUCH_PAGE;
        Graphics2D g2 = (Graphics2D) g;
        g2.translate(pf.getImageableX(), pf.getImageableY());
        drawPage(g2);
        return Printable.PAGE_EXISTS;
    }

    public void drawPage(Graphics2D g2)
    {
        // здесь следует общий код для рисования
        . . .
    }
    . . .
}

```

В рассматриваемом здесь примере для демонстрации вывода данных на печать используется то же самое изображение, что и на рис. 11.20. В этом изображении контуры символов, составляющих строку "Hello, World", используются в качестве области отсечения для линий штрихового рисунка.

Для запуска задания на печать достаточно щелкнуть на кнопке Print (Печать), а для открытия диалогового окна настройки печати — на кнопке Page setup (Параметры страницы). Соответствующий исходный код приведен в листинге 11.10.



На заметку! Чтобы отобразить платформенно-ориентированное диалоговое окно параметров печатаемой страницы, методу `pageDialog()` следует передать объект типа `PageFormat`. Этот метод создает копию объекта, изменяет ее в соответствии с пользовательскими установками и возвращает обратно, как показано ниже.

```

PageFormat defaultFormat = printJob.defaultPage();
PageFormat selectedFormat = printJob.pageDialog(defaultFormat);

```

Листинг 11.10. Исходный код из файла print/PrintTestFrame.java

```

1 package print;
2
3 import java.awt.*;
4 import java.awt.print.*;
5
6 import javax.print.attribute.*;
7 import javax.swing.*;
8
9 /**
10  * В этом фрейме отображается панель с двухмерной графикой и
11  * кнопками для печати графики и установки формата страницы
12 */
13 public class PrintTestFrame extends JFrame
14 {

```

```

15 private PrintComponent canvas;
16 private PrintRequestAttributeSet attributes;
17
18 public PrintTestFrame()
19 {
20     canvas = new PrintComponent();
21     add(canvas, BorderLayout.CENTER);
22
23     attributes = new HashPrintRequestAttributeSet();
24
25     JPanel buttonPanel = new JPanel();
26     JButton printButton = new JButton("Print");
27     buttonPanel.add(printButton);
28     printButton.addActionListener(event ->
29     {
30         try
31         {
32             PrinterJob job = PrinterJob.getPrinterJob();
33             job.setPrintable(canvas);
34             if (job.printDialog(attributes)) job.print(attributes);
35         }
36         catch (PrinterException ex)
37         {
38             JOptionPane.showMessageDialog(PrintTestFrame.this, ex);
39         }
40     });
41
42     JButton pageSetupButton = new JButton("Page setup");
43     buttonPanel.add(pageSetupButton);
44     pageSetupButton.addActionListener(event ->
45     {
46         PrinterJob job = PrinterJob.getPrinterJob();
47         job.pageDialog(attributes);
48     });
49
50     add(buttonPanel, BorderLayout.NORTH);
51     pack();
52 }
53 }
```

Листинг 11.11. Исходный код из файла print/PrintComponent.java

```

1 package print;
2
3 import java.awt.*;
4 import java.awt.font.*;
5 import java.awt.geom.*;
6 import java.awt.print.*;
7 import javax.swing.*;
8
9 /**
10  * Этот компонент формирует двухмерную графику для
11  * вывода на экран и на печать
12 */
13 public class PrintComponent extends JComponent implements Printable
14 {
15     private static final Dimension PREFERRED_SIZE =

```

```
16         new Dimension(300, 300);
17     public void paintComponent(Graphics g)
18     {
19         Graphics2D g2 = (Graphics2D) g;
20         drawPage(g2);
21     }
22
23     public int print(Graphics g, PageFormat pf, int page)
24         throws PrinterException
25     {
26         if (page >= 1) return Printable.NO_SUCH_PAGE;
27         Graphics2D g2 = (Graphics2D) g;
28         g2.translate(pf.getImageableX(), pf.getImageableY());
29         g2.draw(new Rectangle2D.Double(0, 0,
30             pf.getImageableWidth(), pf.getImageableHeight()));
31
32         drawPage(g2);
33         return Printable.PAGE_EXISTS;
34     }
35
36     /**
37      * Этот метод рисует страницу в графическом контексте
38      * экрана и принтера
39      * @param g2 Графический контекст
40      */
41     public void drawPage(Graphics2D g2)
42     {
43         FontRenderContext context = g2.getFontRenderContext();
44         Font f = new Font("Serif", Font.PLAIN, 72);
45         GeneralPath clipShape = new GeneralPath();
46
47         TextLayout layout = new TextLayout("Hello", f, context);
48         AffineTransform transform =
49             AffineTransform.getTranslateInstance(0, 72);
50         Shape outline = layout.getOutline(transform);
51         clipShape.append(outline, false);
52
53         layout = new TextLayout("World", f, context);
54         transform = AffineTransform.getTranslateInstance(0, 144);
55         outline = layout.getOutline(transform);
56         clipShape.append(outline, false);
57
58         g2.draw(clipShape);
59         g2.clip(clipShape);
60
61         final int NLINES = 50;
62         Point2D p = new Point2D.Double(0, 0);
63         for (int i = 0; i < NLINES; i++)
64         {
65             double x = (2 * getWidth() * i) / NLINES;
66             double y = (2 * getHeight() * (NLINES - 1 - i)) / NLINES;
67             Point2D q = new Point2D.Double(x, y);
68             g2.draw(new Line2D.Double(p, q));
69         }
70     }
71
72     public Dimension getPreferredSize() { return PREFERRED_SIZE; }
73 }
```

java.awt.print.Printable 1.2

- **int print(Graphics g, PageFormat format, int pageNumber)**

Воспроизводит страницу и возвращает значение **PAGE_EXISTS** или **NO_SUCH_PAGE**.

Параметры: **g** Графический контекст, в котором

воспроизводится страница

format Формат воспроизводимой страницы

pageNumber Номер требуемой страницы

java.awt.print.PrinterJob 1.2

- **static PrinterJob getPrinterJob()**

Возвращает объект задания на печать.

- **PageFormat defaultPage()**

Возвращает установленный по умолчанию формат страницы для данного печатающего устройства.

- **boolean printDialog(PrintRequestAttributeSet attributes)**

- **boolean printDialog()**

Открывают диалоговое окно, где пользователь может выбирать страницы для печати и настраивать ее параметры. Первый метод отображает платформенно-независимое диалоговое окно, а второй — платформенно-ориентированное окно. Оба метода изменяют заданный объект **attributes** с учетом пользовательских настроек и возвращают логическое значение **true**, если пользователь подтвердит установленные параметры настройки печати.

- **PageFormat pageDialog(PrintRequestAttributeSet attributes)**

- **PageFormat pageDialog(PageFormat defaults)**

Вызывают диалоговое окно настройки страницы. Первый из этих методов отображает платформенно-независимое диалоговое окно, а второй — платформенно-ориентированное окно. Оба метода возвращают объект типа **PageFormat**, определяющий формат, указанный пользователем в диалоговом окне. Первый метод видоизменяет заданный объект **attributes** таким образом, чтобы тот отражал пользовательские установки. А второй метод не видоизменяет объект **defaults**.

- **void setPrintable(Printable p)**

- **void setPrintable(Printable p, PageFormat format)**

Устанавливают объект типа **Printable** и необязательный формат страницы для текущего задания на печать.

- **void print()**

- **void print(PrintRequestAttributeSet attributes)**

Вызывают на печать текущий объект типа **Printable**, повторно вызывая его метод **print()** и отправляя воспроизводимые страницы на печатающее устройство до тех пор, пока не будут напечатаны все страницы.

java.awt.print.PageFormat 1.2

- **double getWidth()**
- **double getHeight()**
Возвращают ширину и высоту страницы.
- **double getImageableWidth()**
- **double getImageableHeight()**
Возвращают ширину и высоту доступной для печати области страницы.
- **double getImageableX()**
- **double getImageableY()**
Возвращают координаты верхнего левого угла доступной для печати области страницы.
- **int getOrientation()**
Возвращают значение одной из следующих констант, определяющих ориентацию страницы: **PORTRAIT**, **LANDSCAPE** или **REVERSE LANDSCAPE**. Ориентация прозрачна для программирования, поскольку она автоматически учитывается в установках формата страниц и графического контекста.

11.12.2. Многостраничная печать

На практике объекты типа **Printable** нежелательно передавать заданию на печать без предварительной обработки. Сначала следует получить экземпляр класса, реализующего интерфейс **Pageable**. На платформе Java для этой цели предоставляется класс **Book**, реализующий книгу, состоящую из разделов, представленных объектами типа **Printable**. Для составления книги сначала вводятся объекты типа **Printable** в ее разделы и организуется нумерация страниц, как показано ниже.

```
Book book = new Book();
Printable coverPage = . . . ;
Printable bodyPages = . . . ;
book.append(coverPage, pageFormat); // присоединить одну страницу
book.append(bodyPages, pageFormat, pageCount);
```

Затем вызывается метод **setPageable()**, чтобы передать книгу в виде объекта типа **Book** заданию на печать:

```
printJob.setPageable(book);
```

Теперь заданию на печать точно известно количество печатаемых страниц, а в диалоговом окне настройки печати отображаются их номера, чтобы пользователь мог выбрать весь диапазон печатаемых страниц или только его часть.



Внимание! Когда из задания на печать вызывается метод **print()** для объектов разделов типа **Printable**, ему передается текущий номер страницы, отсчитываемый в пределах книги, а не каждого раздела. Это очень неудобно, поскольку в каждом разделе должны храниться данные о количестве страниц в предыдущих разделах.

Для программирования наибольшую сложность при использовании класса **Book** представляет подсчет и сохранение данных о количестве страниц в каждом разделе во время печати. В объекте типа **Printable** следует создать алгоритм

компоновки печатаемой страницы. Компоновку страницы необходимо рассчитать перед началом ее печати, чтобы определить количество печатаемых страниц и их разрывы. Сведения о компоновке страницы можно сохранить для дальнейшего применения во время печати. Необходимо также учитывать, что пользователь может изменить формат страницы. В таком случае придется повторно рассчитать компоновку страницы, даже если печатаемые данные не изменились.

В примере программы из листинга 11.13 показано, каким образом организуется многостраничная печать. Эта программа выводит крупный заголовок (так называемый *баннер*), состоящий из строки символов очень большого размера, и поэтому они располагаются на нескольких страницах (рис. 11.35). После распечатки всех страниц поля обрезаются, а страницы склеиваются вместе.

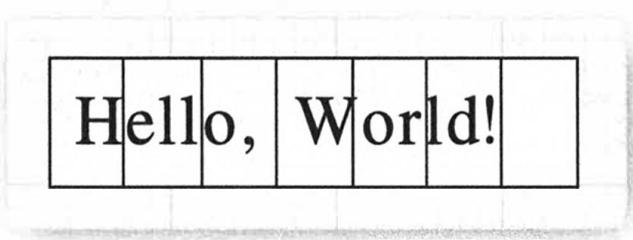


Рис. 11.35. Баннер

Метод `layoutPages()` из класса `Banner` рассчитывает компоновку страницы. Сначала компонуется строка сообщения, отформатированная шрифтом 72 пункта. Затем высота результирующей строки вычисляется и сравнивается с высотой печатаемой области страницы. На основании этих двух измерений определяется масштабный коэффициент. При выводе на печать строка увеличивается на этот масштабный коэффициент.



Внимание! Для точного размещения на странице печатаемой информации требуется доступ к графическому контексту печатающего устройства. К сожалению, получить доступ к графическому контексту нельзя до тех пор, пока не начнется сама печать. В программе, рассматриваемой здесь в качестве примера, все требующиеся действия выполняются в экранном графическом контексте в надежде, что типографские параметры шрифтов на экране и на печатной странице совпадут.

В методе `getPageCount()` из класса `Banner` сначала вызывается метод определения компоновки страницы. Затем ширина символьной строки увеличивается в масштабе и делится на ширину доступной для печати области отдельной страницы. Частное от этого деления, округленное до следующего целого значения, определяет искомое количество страниц.

Из всего сказанного выше можно сделать вывод, что напечатать баннер не так-то просто, поскольку отдельные части символа могут располагаться на разных страницах. Но благодаря эффективным средствам Java 2D API эта задача решается очень просто. Для получения конкретной страницы вызывается метод `translate()` из класса `Graphics2D`, смещающий верхний левый угол печатаемой символьной строки на заданное расстояние влево. Затем устанавливается

прямоугольная область отсечения с размерами текущей страницы (рис. 11.36). И наконец, масштаб графического контекста изменяется по масштабному коэффициенту, вычисленному методом компоновки страницы.

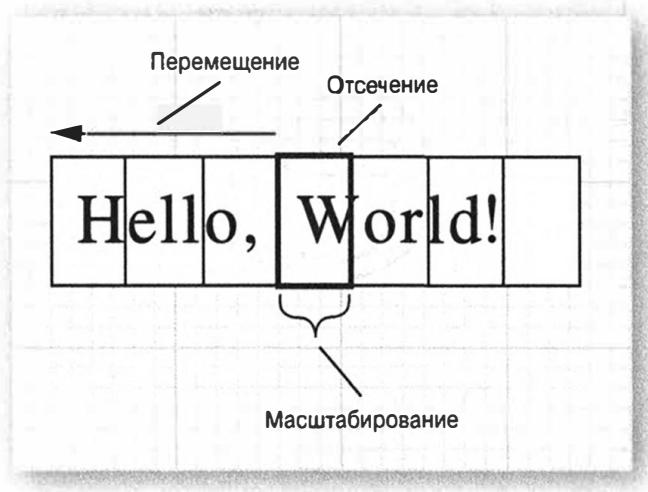


Рис. 11.36. Алгоритм постраничной печати баннера

Рассматриваемый здесь пример демонстрирует истинный потенциал преобразований, благодаря которым код для воспроизведения графического изображения на странице остается простым. Как видите, преобразования выполняют всю работу по размещению изображения в соответствующем месте и отсечению той его части, которая выходит за пределы доступной для печати области страницы. В следующем разделе рассматривается еще один наглядный пример применения преобразований для организации предварительного просмотра печатаемой страницы.

11.12.3. Предварительный просмотр печати

В большинстве прикладных программ профессионального уровня предусмотрен механизм предварительного просмотра, который позволяет оценить внешний вид страницы на экране компьютера перед запуском задания на печать, чтобы не тратить зря бумагу на распечатку неверно скомпонованных страниц. В классах платформы Java, предназначенных для вывода данных на печать, не предусмотрено стандартное диалоговое окно предварительного просмотра печати. Тем не менее окно, аналогичное приведенному на рис. 11.37, можно без особых трудов создать самостоятельно. В этом разделе рассматривается один из способов создания такого окна. В частности, класс `PrintPreviewDialog` из листинга 11.14 можно рассматривать как полностью обобщенный, поскольку он обеспечивает предварительный просмотр любого вида печати.

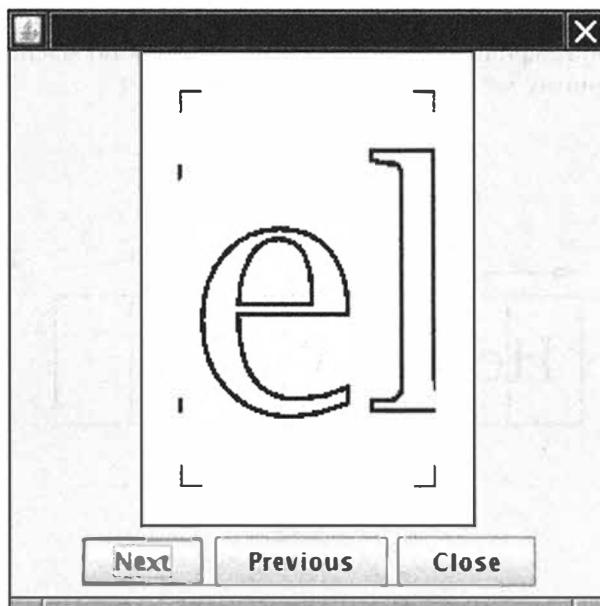


Рис. 11.37. Диалоговое окно предварительного просмотра печатаемой страницы с баннером

Для создания диалогового окна предварительного просмотра печати конструируется объект типа `PrintPreviewDialog` из объектов типа `Printable` или `Book`, а также объекта типа `PageFormat`. А для оформления внешнего вида диалогового окна служит класс `PrintPreviewCanvas` из листинга 11.15. При активизации кнопок `Next` и `Previous`, предназначенных для перехода к следующей и предыдущей просматриваемым страницам соответственно, из метода `paintComponent()` вызывается метод `print()` для объекта типа `Printable` запрашиваемой страницы.

Обычно метод `print()` отображает содержимое печатаемой страницы в графическом контексте печатающего устройства. Но в данном случае используется графический контекст экрана, масштабированный таким образом, чтобы печатаемая страница полностью вмещалась в небольшой прямоугольной области экрана, как показано ниже.

```
float xoff = . . .; // левая сторона страницы
float yoff = . . .; // верхняя сторона страницы
float scale = . . .; // масштабный коэффициент для подгонки
    // печатаемой страницы по размерам прямоугольной области экрана
g2.translate(xoff, yoff);
g2.scale(scale, scale);
Printable printable = book.getPrintable(currentPage);
printable.print(g2, pageFormat, currentPage);
```

В данном случае метод `print()` выполняет те же функции, что и при выводе страницы на печать, просто рисуя в заданном графическом контексте страницу, которая отображается на экране в уменьшенном виде для предварительного просмотра. Данный пример наглядно демонстрирует возможности модели воспроизведения изображений в Java 2D API.

В листинге 11.12 представлен исходный код примера программы для вывода баннера на печать с диалоговым окном предварительного просмотра. Эта программа позволяет ввести строку "Hello, World!" в текстовом поле, оценить внешний вид печатаемой страницы, а затем напечатать введенную строку в виде баннера.

Листинг 11.12. Исходный код из файла Book/BookTestFrame.java

```
1 package book;
2
3 import java.awt.*;
4 import java.awt.print.*;
5
6 import javax.print.attribute.*;
7 import javax.swing.*;
8
9 /**
10  * Этот фрейм содержит текстовое поле для ввода баннера,
11  * а также кнопки для печати, настройки и предварительного
12  * просмотра печатаемой страницы
13 */
14 public class BookTestFrame extends JFrame
15 {
16     private JTextField text;
17     private PageFormat pageFormat;
18     private PrintRequestAttributeSet attributes;
19
20     public BookTestFrame()
21     {
22         text = new JTextField();
23         add(text, BorderLayout.NORTH);
24
25         attributes = new HashPrintRequestAttributeSet();
26
27         JPanel buttonPanel = new JPanel();
28
29         JButton printButton = new JButton("Print");
30         buttonPanel.add(printButton);
31         printButton.addActionListener(event ->
32         {
33             try
34             {
35                 PrinterJob job = PrinterJob.getPrinterJob();
36                 job.setPageable(makeBook());
37                 if (job.printDialog(attributes))
38                 {
39                     job.print(attributes);
40                 }
41             }
42             catch (PrinterException e)
43             {
44                 JOptionPane.showMessageDialog(BookTestFrame.this, e);
45             }
46         });
47
48     JButton pageSetupButton = new JButton("Page setup");
```

```

49     buttonPanel.add(pageSetupButton);
50     pageSetupButton.addActionListener(event ->
51     {
52         PrinterJob job = PrinterJob.getPrinterJob();
53         pageFormat = job.pageDialog(attributes);
54     });
55
56     JButton printPreviewButton = new JButton("Print preview");
57     buttonPanel.add(printPreviewButton);
58     printPreviewButton.addActionListener(event ->
59     {
60         PrintPreviewDialog dialog =
61                     new PrintPreviewDialog(makeBook());
62         dialog.setVisible(true);
63     });
64
65     add(buttonPanel, BorderLayout.SOUTH);
66     pack();
67 }
68
69 /**
70 * Составляет книгу из страницы обложки и страниц баннера
71 */
72 public Book makeBook()
73 {
74     if (pageFormat == null)
75     {
76         PrinterJob job = PrinterJob.getPrinterJob();
77         pageFormat = job.defaultPage();
78     }
79     Book book = new Book();
80     String message = text.getText();
81     Banner banner = new Banner(message);
82     int pageCount = banner.getPageCount(
83             (Graphics2D) getGraphics(), pageFormat);
84     book.append(new CoverPage(
85             message + " (" + pageCount + " pages)", pageFormat));
86     book.append(banner, pageFormat, pageCount);
87     return book;
88 }
89 }
```

Листинг 11.13. Исходный код из файла book/Banner.java

```

1 package book;
2
3 import java.awt.*;
4 import java.awt.font.*;
5 import java.awt.geom.*;
6 import java.awt.print.*;
7
8 /**
9  * Баннер для печати текстовой строки на нескольких страницах
10 */
11 public class Banner implements Printable
12 {
13     private String message;
```

```
14 private double scale;
15
16 /**
17 * Конструирует баннер
18 * @param m Стока сообщения
19 */
20 public Banner(String m)
21 {
22     message = m;
23 }
24
25 /**
26 * Получает количество страниц в данном разделе
27 * @param g2 Графический контекст
28 * @param pf Формат страницы
29 * @return Возвращает количество требующихся страниц
30 */
31 public int getPageCount(Graphics2D g2, PageFormat pf)
32 {
33     if (message.equals("")) return 0;
34     FontRenderContext context = g2.getFontRenderContext();
35     Font f = new Font("Serif", Font.PLAIN, 72);
36     Rectangle2D bounds = f.getStringBounds(message, context);
37     scale = pf.getImageableHeight() / bounds.getHeight();
38     double width = scale * bounds.getWidth();
39     int pages = (int) Math.ceil(width / pf.getImageableWidth());
40     return pages;
41 }
42
43 public int print(Graphics g, PageFormat pf, int page)
44     throws PrinterException
45 {
46     Graphics2D g2 = (Graphics2D) g;
47     if (page > getPageCount(g2, pf))
48         return Printable.NO_SUCH_PAGE;
49     g2.translate(pf.getImageableX(), pf.getImageableY());
50
51     drawPage(g2, pf, page);
52     return Printable.PAGE_EXISTS;
53 }
54
55 public void drawPage(Graphics2D g2, PageFormat pf, int page)
56 {
57     if (message.equals("")) return;
58     page--; // учитывать страницу обложки
59
60     drawCropMarks(g2, pf);
61     g2.clip(new Rectangle2D.Double(0, 0, pf.getImageableWidth(),
62                                   pf.getImageableHeight()));
63     g2.translate(-page * pf.getImageableWidth(), 0);
64     g2.scale(scale, scale);
65     FontRenderContext context = g2.getFontRenderContext();
66     Font f = new Font("Serif", Font.PLAIN, 72);
67     TextLayout layout = new TextLayout(message, f, context);
68     AffineTransform transform = AffineTransform
69         .getTranslateInstance(0, layout.getAscent());
70     Shape outline = layout.getOutline(transform);
71     g2.draw(outline);
```

```
72     }
73
74     /**
75      * Рисует полудюймовые метки обрезки в углах страницы
76      * @param g2 Графический контекст
77      * @param pf Формат страницы
78     */
79    public void drawCropMarks(Graphics2D g2, PageFormat pf)
80    {
81        final double C = 36; // длина метки обрезки = 1/2 дюйма
82        double w = pf.getImageableWidth();
83        double h = pf.getImageableHeight();
84        g2.draw(new Line2D.Double(0, 0, 0, C));
85        g2.draw(new Line2D.Double(0, 0, C, 0));
86        g2.draw(new Line2D.Double(w, 0, w, C));
87        g2.draw(new Line2D.Double(w, 0, w - C, 0));
88        g2.draw(new Line2D.Double(0, h, 0, h - C));
89        g2.draw(new Line2D.Double(0, h, C, h));
90        g2.draw(new Line2D.Double(w, h, w, h - C));
91        g2.draw(new Line2D.Double(w, h, w - C, h));
92    }
93 }
94
95 /**
96  * Этот класс печатает страницу обложки с заглавием
97 */
98 class CoverPage implements Printable
99 {
100    private String title;
101
102    /**
103     * Конструирует страницу обложки
104     * @param t Заглавие
105    */
106    public CoverPage(String t)
107    {
108        title = t;
109    }
110
111    public int print(Graphics g, PageFormat pf, int page)
112        throws PrinterException
113    {
114        if (page >= 1) return Printable.NO_SUCH_PAGE;
115        Graphics2D g2 = (Graphics2D) g;
116        g2.setPaint(Color.black);
117        g2.translate(pf.getImageableX(), pf.getImageableY());
118        FontRenderContext context = g2.getFontRenderContext();
119        Font f = g2.getFont();
120        TextLayout layout = new TextLayout(title, f, context);
121        float ascent = layout.getAscent();
122        g2.drawString(title, 0, ascent);
123        return Printable.PAGE_EXISTS;
124    }
125 }
```

Листинг 11.14. Исходный код из файла book/PrintPreviewDialog.java

```
1 package book;
2
3 import java.awt.*;
4 import java.awt.print.*;
5
6 import javax.swing.*;
7
8 /**
9  * Этот класс реализует типичное окно предварительного
10 * просмотра печати
11 */
12 public class PrintPreviewDialog extends JDialog
13 {
14     private static final int DEFAULT_WIDTH = 300;
15     private static final int DEFAULT_HEIGHT = 300;
16
17     private PrintPreviewCanvas canvas;
18
19 /**
20  * Конструирует диалоговое окно предварительного
21  * просмотра печати
22  * @param p Печатаемый объект типа Printable
23  * @param pf Формат страницы
24  * @param pages Количество страниц в печатаемом объекте p
25  */
26     public PrintPreviewDialog(Printable p, PageFormat pf, int pages)
27     {
28         Book book = new Book();
29         book.append(p, pf, pages);
30         layoutUI(book);
31     }
32
33 /**
34  * Создает диалоговое окно предварительного просмотра печати
35  * @param b Объект книги типа Book
36  */
37     public PrintPreviewDialog(Book b)
38     {
39         layoutUI(b);
40     }
41
42 /**
43  * Компонует ГПИ в диалоговом окне
44  * @param book Предварительно просматриваемая книга
45  */
46     public void layoutUI(Book book)
47     {
48         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
49
50         canvas = new PrintPreviewCanvas(book);
51         add(canvas, BorderLayout.CENTER);
52
53         JPanel buttonPanel = new JPanel();
54         JButton nextButton = new JButton("Next");
55     }
56 }
```

```
56     buttonPanel.add(nextButton);
57     nextButton.addActionListener(event -> canvas.flipPage(1));
58
59     JButton previousButton = new JButton("Previous");
60     buttonPanel.add(previousButton);
61     previousButton.addActionListener(event ->
62                                     canvas.flipPage(-1));
63
64     JButton closeButton = new JButton("Close");
65     buttonPanel.add(closeButton);
66     closeButton.addActionListener(event -> setVisible(false));
67
68     add(buttonPanel, BorderLayout.SOUTH);
69 }
70 }
```

Листинг 11.15. Исходный код из файла book/PrintPreviewCanvas.java

```
1 package book;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import java.awt.print.*;
6 import javax.swing.*;
7
8 /**
9  * Канва для отображения предварительного просмотра печати
10 */
11 class PrintPreviewCanvas extends JComponent
12 {
13     private Book book;
14     private int currentPage;
15
16     /**
17      * Конструирует канву для предварительного просмотра печати
18      * @param b Предварительно просматриваемая книга
19     */
20     public PrintPreviewCanvas(Book b)
21     {
22         book = b;
23         currentPage = 0;
24     }
25     public void paintComponent(Graphics g)
26     {
27         Graphics2D g2 = (Graphics2D) g;
28         PageFormat pageFormat = book.getPageFormat(currentPage);
29
30         double xoff; // координата x смещения начала страницы в окне
31         double yoff; // координата y смещения начала страницы в окне
32         double scale; // масштабный коэффициент для подгонки
33                     // просматриваемой страницы по размерам окна
34         double px = pageFormat.getWidth();
35         double py = pageFormat.getHeight();
36         double sx = getWidth() - 1;
37         double sy = getHeight() - 1;
38         if (px / py < sx / sy) // отцентрововать по горизонтали
39     }
```

```

40     scale = sy / py;
41     xoff = 0.5 * (sx - scale * px);
42     yoff = 0;
43 }
44 else
45 // отцентровать по вертикали
46 {
47     scale = sx / px;
48     xoff = 0;
49     yoff = 0.5 * (sy - scale * py);
50 }
51 g2.translate((float) xoff, (float) yoff);
52 g2.scale((float) scale, (float) scale);
53
54 // нарисовать контуры страницы page, игнорируя поля
55 Rectangle2D page = new Rectangle2D.Double(0, 0, px, py);
56 g2.setPaint(Color.white);
57 g2.fill(page);
58 g2.setPaint(Color.black);
59 g2.draw(page);
60
61 Printable printable = book.getPrintable(currentPage);
62 try
63 {
64     printable.print(g2, pageFormat, currentPage);
65 }
66 catch (PrinterException e)
67 {
68     g2.draw(new Line2D.Double(0, 0, px, py));
69     g2.draw(new Line2D.Double(px, 0, 0, py));
70 }
71 }
72 /**
73 * Листать книгу на заданное число страниц
74 * @param by Число листаемых страниц. Отрицательные значения
75 *           данного параметра обозначают листание книги назад
76 */
77 public void flipPage(int by)
78 {
79     int nextPage = currentPage + by;
80     if (0 <= nextPage && nextPage < book.getNumberOfPages())
81     {
82         currentPage = nextPage;
83         repaint();
84     }
85 }
86 }
```

java.awt.print.PrinterJob 1.2

- **void setPageable(Pageable p)**

Устанавливает печатаемый объект типа **Pageable**, например, объект книги типа **Book**.

java.awt.print.Book 1.2

- **void append(Printable p, PageFormat format)**
- **void append(Printable p, PageFormat format, int pageCount)**
Добавляют новый раздел в данную книгу. Если количество страниц не указано, то добавляется первая страница.
- **Printable getPrintable(int page)**
Получает печатаемый объект типа **Printable** для указанной страницы.

11.12.4. Службы печати

В предыдущих разделах было показано, как выводить двухмерную графику на печать. Но прикладной программный интерфейс API для печати, внедренный в версии Java SE 1.4, обладает более широкими возможностями. В этом прикладном интерфейсе определен целый ряд типов данных и поддерживается поиск служб печати для их вывода. Ниже перечислены некоторые из поддерживаемых типов данных.

- Изображения в формате GIF, JPEG или PNG.
- Текстовые документы в формате HTML, PostScript или PDF.
- Неформатированные данные для печатающего устройства.
- Экземпляры класса, реализующего интерфейсы **Printable**, **Pageable** или **RenderableImage**.

Сами данные могут храниться в любом источнике байтов или символов, например, в потоке ввода, веб-ресурсе, доступном по указанному URL, или массиве. Сочетание типа данных и источника данных называется *разновидностью документа*. Так, в классе **DocFlavor** определен целый ряд внутренних классов для различных источников данных. Каждый из этих внутренних классов содержит константы для указания разновидности документа. Например, приведенная ниже константа описывает изображение формата GIF, считываемое из потока ввода. Все разновидности документов, т.е. допустимые сочетания типов источников и данных, включая и тип MIME, перечислены в табл. 11.3.

DocFlavor.INPUT_STREAM.GIF

Таблица 11.3. Разновидности документов для служб печати

Источник данных	Тип данных	Тип MIME
INPUT_STREAM	GIF	image/gif
URL	JPEG	image/jpeg
BYTE_ARRAY	PNG	image/png
	POSTSCRIPT	application/postscript
	PDF	application/pdf
TEXT_HTML_HOST		text/html (в кодировке, характерной для конкретного узла)
TEXT_HTML_US_ASCII		text/html; charset=us-ascii
TEXT_HTML_UTF_8		text/html; charset=utf-8

Окончание табл. 11.3

Источник данных	Тип данных	Тип MIME
	TEXT_HTML_UTF_16	text/html; charset=utf-16
	TEXT_HTML_UTF_16LE	text/html; charset=utf-16le (прямой порядок следования байтов)
	TEXT_HTML_UTF_16BE	text/html; charset=utf-16be (обратный порядок следования байтов)
	TEXT_PLAIN_HOST	text/plain (в кодировке, характерной для конкретного узла)
	TEXT_PLAIN_US_ASCII	text/plain; charset=us-ascii
	TEXT_PLAIN_UTF_8	text/plain; charset=utf-8
	TEXT_PLAIN_UTF_16	text/plain; charset=utf-16
	TEXT_PLAIN_UTF_16LE	text/plain; charset=utf-16le (прямой порядок следования байтов)
	TEXT_PLAIN_UTF_16BE	text/plain; charset=utf-16be (обратный порядок следования байтов)
	PCL	application/vnd.hp-PCL (Hewlett Packard Printer Control Language)
	AUTOSENSE	application/octet-stream (неформатированные данные для печатающего устройства)
READER	TEXT_HTML	text/html; charset=utf-16
STRING	TEXT_PLAIN	text/plain; charset=utf-16
CHAR_ARRAY		
SERVICE_FORMATTED	PRINTABLE	Отсутствует
	PAGEABLE	Отсутствует
	RENDERABLE_IMAGE	Отсутствует

Допустим, требуется напечатать изображение из файла формата GIF. Для этого нужно сначала выяснить, имеется ли соответствующая служба печати, способная справиться с подобной задачей. Статический метод `lookupPrintServices()` из класса `PrintServiceLookup` возвращает массив объектов типа `PrintService`, способных напечатать указанную разновидность документа, как показано ниже.

```
DocFlavor flavor = DocFlavor.INPUT_STREAM.GIF;
PrintService[] services =
    PrintServiceLookup.lookupPrintServices(flavor, null);
```

Второй параметр метода `lookupPrintServices()` принимает пустое значение `null`, которое означает, что поиск подходящей службы печати не ограничивается какими-то определенными атрибутами печати. Более подробно атрибуты печати рассматриваются в следующем разделе.

Если в результате поиска служб печати получен массив `PrintService[]` с несколькими элементами, то среди перечисленных в нем служб печати следует выбирать какую-нибудь одну. Чтобы извлечь список имен служб печати (например, имен печатающих устройств) и предоставить пользователю возможность выбора, достаточно вызвать метод `getName()` из класса `PrintService`.

Далее из выбранной службы следует получить задание на печать:

```
DocPrintJob job = services[i].createPrintJob();
```

Для печати документа определенной разновидности необходимо создать объект, класс которого реализует интерфейс `Doc`. Для этой цели в библиотеке Java предусмотрен класс `SimpleDoc`. Конструктору класса `SimpleDoc` следует передать объект, представляющий источник данных, разновидность документа и дополнительный, но необязательный набор атрибутов печати, как показано в приведенном ниже примере кода.

```
InputStream in = new FileInputStream(fileName);
Doc doc = new SimpleDoc(in, flavor, null);
```

И наконец, когда все готово для печати, можно приступить к выводу документа на печать, вызвав приведенный ниже метод. Как и прежде, пустое значение `null` второго параметра этого метода можно заменить набором атрибутов печати.

```
job.print(doc, null);
```

Следует, однако, иметь в виду, что описанный выше процесс печати совершен но не похож на рассматривавшийся в предыдущем разделе, потому что в данном случае отсутствует всякое взаимодействие с пользователем посредством диалоговых окон. Применяя подобный механизм, можно организовать печать на сервере, где пользователи должны передавать задания на печать из специальной веб-формы. В примере программы, исходный код которой приведен в листинге 11.16, демонстрируется применение службы печати для распечатки файла изображения.

Листинг 11.16. Исходный код из файла printService/PrintServiceTest.java

```
1 package printService;
2
3 import java.io.*;
4 import java.nio.file.*;
5 import javax.print.*;
6
7 /**
8  * В этой программе демонстрируется применение служб печати.
9  * Она позволяет напечатать изображение формата GIF в любой
10 * службе печати, поддерживающей разновидность документов
11 * формата GIF
12 * @version 1.10 2007-08-16
13 * @author Cay Horstmann
14 */
15 public class PrintServiceTest
16 {
17     public static void main(String[] args)
18     {
19         DocFlavor flavor = DocFlavor.URL.GIF;
20         PrintService[] services =
21             PrintServiceLookup.lookupPrintServices(flavor, null);
22         if (args.length == 0)
23         {
24             if (services.length == 0) System.out.println(
25                 "No printer for flavor " + flavor);
26             else
27             {
28                 System.out.println("Specify a file of flavor " + flavor
29                     + "\nand optionally the number of the desired printer.");
30                 for (int i = 0; i < services.length; i++)
31                     System.out.println(
32                         (i + 1) + ":" + services[i].getName());
```

```

33     }
34     System.exit(0);
35   }
36   String fileName = args[0];
37   int p = 1;
38   if (args.length > 1) p = Integer.parseInt(args[1]);
39   if (fileName == null) return;
40   try (InputStream in =
41       Files.newInputStream(Paths.get(fileName)))
42   {
43     Doc doc = new SimpleDoc(in, flavor, null);
44     DocPrintJob job = services[p - 1].createPrintJob();
45     job.print(doc, null);
46   }
47   catch (Exception ex)
48   {
49     ex.printStackTrace();
50   }
51 }
52 }
```

javax.print.PrintServiceLookup 1.4

- PrintService[] lookupPrintServices(DocFlavor flavor, AttributeSet attributes)**

Ищет службы печати, способные справиться с указанной разновидностью документа и атрибутами печати.

Параметры:	flavor	Разновидность документа
	attributes	Требуемые атрибуты печати или пустое значение null , если эти атрибуты не принимаются во внимание

javax.print.PrintService 1.4

- DocPrintJob createPrintJob()**

Составляет задание на печать для вывода экземпляра класса, реализующего интерфейс **Doc** (например, **SimpleDoc**).

javax.print.DocPrintJob 1.4

- void print(Doc doc, PrintRequestAttributeSet attributes)**

Выводит на печать заданный документ с указанными атрибутами печати.

Параметры:	doc	Печатаемый документ
	attributes	Требуемые атрибуты печати или пустое значение null , если эти атрибуты не принимаются во внимание

javax.print.SimpleDoc 1.4

- **SimpleDoc(Object data, DocFlavor flavor, DocAttributeSet attributes)**
Создает объект типа **SimpleDoc**, который может быть выведен на печать в задании типа **DocPrintJob**.

Параметры:

dataОбъект с данными для печати,
например, поток ввода или объект**flavor**типа **Printable**Разновидность документа с данными
для печати**attributes**Атрибуты печати документа или
пустое значение **null**, если
атрибуты печати не требуются

11.12.5. Потоковые службы печати

Обычные службы печати направляют данные на печатающее устройство. А потоковые службы печати формируют данные аналогичным образом, но направляют их в поток вывода. Такая потребность возникает в тех случаях, когда необходимо задержать печать или интерпретировать формат печати данных в других программах. Если, например, данные печатаются в формате PostScript, их целесообразно сохранить в файле, поскольку многие программы способны обращаться с файлами формата PostScript. На платформе Java предусмотрена потоковая служба печати, способная распечатывать изображения и двухмерную графику в формате PostScript. Эти службы можно использовать во всех системах — даже в тех, где нет локальных печатающих устройств.

Перечисление потоковых служб печати организовано немного сложнее, чем локальных служб печати. Для этого необходимо получить сначала разновидность типа **DocFlavor** печатаемого объекта и тип MIME потока вывода, а затем массив фабричных объектов, представляющих потоковые службы печати, как показано ниже.

```
DocFlavor flavor = DocFlavor.SERVICE_FORMATTED.PRINTABLE;
String mimeType = "application/postscript";
StreamPrintServiceFactory[] factories = StreamPrintServiceFactory
    .lookupStreamPrintServiceFactories(flavor, mimeType);
```

В классе **StreamPrintServiceFactory** нет ни одного метода, с помощью которого можно было бы отличить один фабричный объект потоковой службы от другого, поэтому выбирается самый первый объект, находящийся в элементе массива **factories[0]**. Для получения объекта потоковой службы печати типа **StreamPrintService** следует вызвать приведенный ниже метод **getPrintService()**, указав поток вывода в качестве его параметра. Класс **StreamPrintService** является производным от класса **PrintService**, поэтому для вывода на печать остается лишь выполнить действия, описанные в предыдущем разделе.

```
OutputStream out = new FileOutputStream(fileName);
StreamPrintService service = factories[0].getPrintService(out);
```

javax.print.StreamPrintServiceFactory 1.4

- **StreamPrintServiceFactory[] lookupStreamPrintServiceFactories(DocFlavor or flavor, String mimeType)**
Ищет потоковые службы печати, способные напечатать документ указанной разновидности и создать поток вывода заданного типа MIME.
- **StreamPrintService getPrintService(OutputStream out)**
Получает службу печати, направляющую данные в указанный поток вывода.

11.12.6. Атрибуты печати

Прикладной программный интерфейс API службы печати содержит сложный набор интерфейсов и классов для указания различных типов атрибутов печати, которые делятся на четыре основные группы. Первые две группы атрибутов печати определяют запросы, направляемые на печатающее устройство.

- *Атрибуты запроса печати.* Запрашивают отдельные характеристики всех документов в задании на печать. В качестве примеров таких характеристик можно привести двухстороннюю печать или формат бумаги.
- *Атрибуты документа.* Используются только для какого-нибудь одного объекта-документа.

Оставшиеся две группы атрибутов содержат сведения о печатающем устройстве и состоянии задания на печать.

- *Атрибуты службы печати.* Содержат сведения о службе печати (например, о производителе и модели печатающего устройства, а также о том, принимает ли печатающее устройство задание на печать в настоящий момент).
- *Атрибуты задания на печать.* Содержат сведения о состоянии определенного задания на печать (например, завершено ли задание или еще не выполнено).

Для описания различных атрибутов печати предусмотрен интерфейс *Attribute* и перечисленные ниже подчиненные интерфейсы.

PrintRequestAttribute
DocAttribute
PrintServiceAttribute
PrintJobAttribute
SupportedValuesAttribute

Отдельные классы атрибутов реализуют один или несколько этих интерфейсов. Например, экземпляры класса *Copies* используются для описания количества копий печатаемого документа. Этот класс реализует интерфейсы *PrintRequestAttribute* и *PrintJobAttribute*. Задание на печать может содержать запрос на печать нескольких копий, тогда как атрибутом задания на печать может быть количество фактически напечатанных копий. Фактическое количество копий может оказаться меньше требуемого из-за ограничений, накладываемых печатающим устройством, или исчерпания бумаги.

Интерфейс `SupportedValuesAttribute` указывает, что значения атрибута не отражают фактический запрос или текущее состояние, а демонстрируют функциональные возможности службы печати. Например, объект класса `CopiesSupported` реализует интерфейс `SupportedValuesAttribute` и сообщает, что печатающее устройство поддерживает печать от 1 до 99 копий. На рис. 11.38 схематически показана иерархия наследования интерфейсов и классов, реализующих атрибуты.

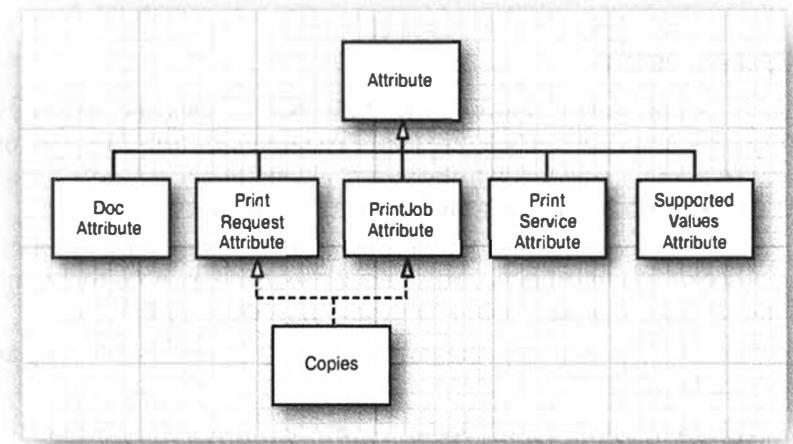


Рис. 11.38. Иерархия наследования интерфейсов и классов, реализующих атрибуты

Помимо интерфейсов и классов для отдельных атрибутов, в прикладном программном интерфейсе API службы печати предусмотрены интерфейсы и классы для наборов атрибутов. Так, у интерфейса `AttributeSet` имеются следующие подчиненные интерфейсы:

`PrintRequestAttributeSet`
`DocAttributeSet`
`PrintServiceAttributeSet`
`PrintJobAttributeSet`

Для каждого из них предусмотрен класс, реализующий свой интерфейс. Все-го насчитывается пять таких классов, перечисленных ниже. На рис. 11.39 схематически показана иерархия наследования интерфейсов и классов, реализующих наборы атрибутов.

`HashAttributeSet`
`HashPrintRequestAttributeSet`
`HashDocAttributeSet`
`HashPrintServiceAttributeSet`
`HashPrintJobAttributeSet`

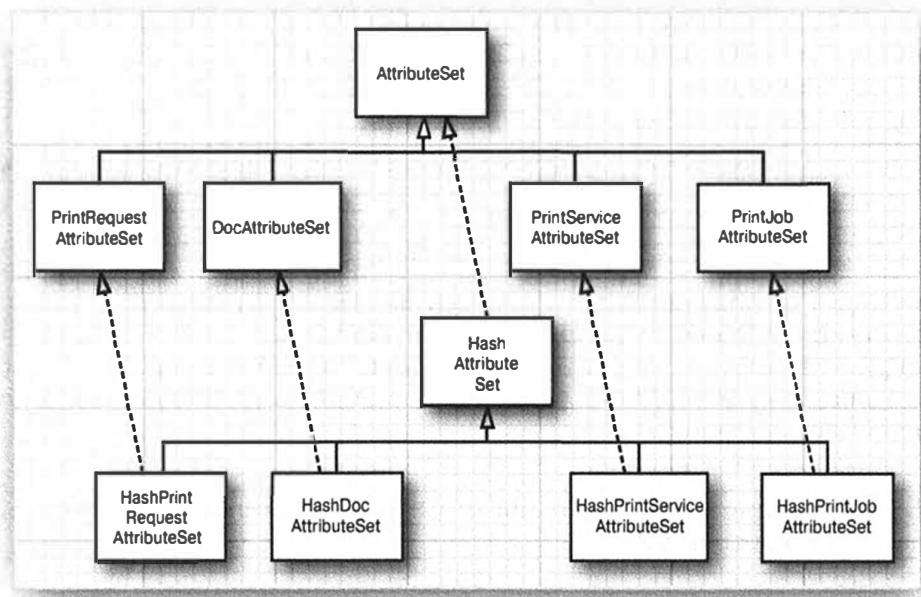


Рис. 11.39. Иерархия наследования интерфейсов и классов, реализующих наборы атрибутов

Например, создать набор атрибутов для запроса на печать можно следующим образом:

```
PrintRequest.AttributeSet attributes = new HashPrintRequest.AttributeSet();
```

После создания набора атрибутов потребность в префиксе `Hash` отпадает. А зачем вообще нужны все эти интерфейсы? Они дают возможность проверить правильность использования атрибутов. Например, интерфейс `DocAttributeSet` принимает только те объекты, классы которых реализуют интерфейс `DocAttribute`, а любая попытка добавить какой-нибудь другой атрибут приведет к ошибке.

Набор атрибутов представляет собой особый вид отображения, где ключи являются объектами класса `Class`, а значения — объектами класса, реализующего интерфейс `Attribute`. Так, если создать и вставить объект в набор атрибутов с помощью операции `new Copies(10)`, то его ключом будет объект `Copies` класса `Class`. Этот ключ называется *категорией* атрибутов. В интерфейсе `Attribute` объявляется приведенный ниже метод, возвращающий категорию атрибута. Класс `Copies` содержит определение этого метода для возврата объекта `Copies.class`, но совсем не обязательно, чтобы категория была объектом того же класса, что и класс атрибута.

```
Class getCategory()
```

При вводе атрибута в набор атрибутов категория извлекается автоматически, поэтому достаточно ввести только значение атрибута, как показано ниже. Если же впоследствии ввести другой атрибут из той же самой категории, он будет записан вместо первого.

```
attributes.add(new Copies(10));
```

Категория используется в качестве ключа для извлечения атрибута, как показано ниже.

```
AttributeSet attributes = job.getAttributes();
Copies copies = (Copies) attribute.get(Copies.class);
```

Атрибуты группируются по типам значений. Например, атрибут типа Copies может иметь любое целочисленное значение, поскольку класс Copies является расширением класса IntegerSyntax, предназначенного для обработки всех целочисленных атрибутов. Метод getValue() из этого класса возвращает целочисленное значение атрибута:

```
int n = copies.getValue();
```

Перечисленные ниже классы инкапсулируют соответственно символьную строку, дату/время или URI.

```
TextSyntax
DateTimeSyntax
URISyntax
```

И наконец, многие атрибуты могут принимать ограниченное количество значений. Например, у атрибута типа PrintQuality имеются всего три допустимых значения для указания экономного, нормального и высокого качества печати, представленных следующими константами:

```
PrintQuality.DRAFT
PrintQuality.NORMAL
PrintQuality.HIGH
```

Классы атрибутов с ограниченным количеством значений расширяют класс EnumSyntax, предоставляющий удобные методы для установки перечислимых значений с учетом типовой безопасности. Наличие таких классов избавляет от необходимости беспокоиться о конкретном механизме реализации при использовании атрибутов. Достаточно ввести в набор атрибутов именованные значения следующим образом:

```
attributes.add(PrintQuality.HIGH);
```

Ниже показано, как проверить значение атрибута.

```
if (attributes.get(PrintQuality.class) == PrintQuality.HIGH)
    . . .
```

Атрибуты печати перечислены в табл. 11.4. Во втором столбце приведен суперкласс каждого класса атрибута (например, класс IntegerSyntax для атрибута типа Copies) или набор перечислимых значений. А в последних четырех столбцах указаны интерфейсы, которые реализует данный класс атрибута: DocAttribute (DA), PrintJobAttribute (PJA), PrintRequestAttribute (PRA) и PrintServiceAttribute (PSA).

Таблица 11.4. Атрибуты печати

Атрибут	Суперкласс или набор констант перечислимого типа	DA	PJA	PRA	PSA
Chromaticity	MONOCHROME, COLOR	✓	✓	✓	
ColorSupported	SUPPORTED, NOT_SUPPORTED				✓
Compression	COMPRESS, DEFLATE, GZIP, NONE		✓		
Copies	IntegerSyntax		✓	✓	
DateTimeAtCompleted	DateTimeSyntax		✓		
DateTimeAtCreation	DateTimeSyntax		✓		
DateTimeAtProcessing	DateTimeSyntax		✓		
Destination	URI Syntax		✓	✓	
DocumentName	TextSyntax		✓		
Fidelity	FIDELITY_TRUE, FIDELITY_FALSE		✓	✓	
Finishings	NONE, STAPLE, EDGE_STITCH, BIND, SADDLE_STITCH, COVER, . . .		✓	✓	✓
JobHoldUntil	DateTimeSyntax		✓	✓	
JobImpressions	IntegerSyntax		✓	✓	
JobImpressionsCompleted	IntegerSyntax		✓		
JobKOctets	IntegerSyntax		✓	✓	
JobKOctetsProcessed	IntegerSyntax		✓		
JobMediaSheets	IntegerSyntax		✓	✓	
JobMediaSheetsCompleted	IntegerSyntax		✓		
JobMessageFromOperator	TextSyntax		✓		
JobName	TextSyntax		✓	✓	
JobOriginatingUserName	TextSyntax		✓		
JobPriority	IntegerSyntax		✓	✓	
JobSheets	STANDARD, NONE		✓	✓	
JobState	ABORTED, CANCELED, COMPLETED, PENDING, PENDING_HELD, PROCESSING, PROCESSING_STOPPED		✓		
JobStateReason	ABORTED_BY_SYSTEM, DOCUMENT_ FORMAT_ERROR и проч.				
JobStateReasons	HashSet			✓	
MediaName	ISO_A4_WHITE, ISO_A4_TRANSPARENT, NA LETTER WHITE, NA LETTER_ TRANSPARENT		✓	✓	✓
MediaSize	ISO.A0-ISO.A10, ISO.B0-ISO. B10, ISO.C0-ISO.C10, NA.LETTER, NA.LEGAL и другие форматы страниц и бумаги				
MediaSizeName	ISO.A0-ISO.A10, ISO.B0-ISO. B10, ISO.C0-ISO.C10, NA.LETTER, NA.LEGAL и другие форматы страниц и бумаги		✓	✓	✓
MediaTray	TOP, MIDDLE, BOTTOM, SIDE, ENVELOPE, LARGE_CAPACITY, MAIN, MANUAL		✓	✓	✓

Окончание табл. 11.4

Атрибут	Суперкласс или набор констант перечислимого типа	DA	PJA	PRA	PSA
MultipleDocumentHandling	SINGLE_DOCUMENT, SINGLE_DOCUMENT_, NEW_SHEET, SEPARATE_DOCUMENTS_, COLLATED_COPIES, SEPARATE_COPIES_, DOCUMENTS_UNCOLLATED_COPIES_	✓	✓		
NumberOfDocuments	IntegerSyntax	✓			
NumberOfInterveningJobs	IntegerSyntax		✓		
NumberUp	IntegerSyntax	✓	✓	✓	
OrientationRequested	PORTRAIT, LANDSCAPE, REVERSE_, PORTRAIT, REVERSE_LANDSCAPE_	✓	✓	✓	
OutputDeviceAssigned	TextSyntax		✓		
PageRanges	SetOfInteger	✓	✓	✓	
PagesPerMinute	IntegerSyntax				✓
PagesPerMinuteColor	IntegerSyntax				✓
PDLOverrideSupported	ATTEMPTED, NOT_ATTEMPTED				✓
PresentationDirection	TORIGHT_TOBOTTOM, TORIGHT_TOTOP, TOBOTTOM_TORIGHT, TOBOTTOM_TOLEFT_, TOLEFT_TOLEFT_TOBOTTOM, TOLEFT_TOTOP_, TOTOP_TOTOP_TORIGHT, TOTOP_TOLEFT		✓	✓	
PrinterInfo	TextSyntax				✓
PrinterIsAcceptingJobs	ACCEPTING_JOBS, NOT_ACCEPTING_JOBS				✓
PrinterLocation	TextSyntax				✓
PrinterMakeAndModel	TextSyntax				✓
PrinterMessageFromOperator	TextSyntax				✓
PrinterMoreInfo	URISyntax				✓
PrinterMoreInfoManufacturer	URISyntax				✓
PrinterName	TextSyntax				✓
PrinterResolution	ResolutionSyntax	✓	✓	✓	
PrinterState	PROCESSING, IDLE, STOPPED, UNKNOWN				✓
PrinterStateReason	COVER_OPEN, FUSER_OVER_TEMP, MEDIA_JAM и проч.				
PrinterStateReasons	HashMap				
PrinterURI	URISyntax				✓
PrintQuality	DRAFT, NORMAL, HIGH	✓	✓	✓	
QueuedJobCount	IntegerSyntax				✓
ReferenceUriSchemesSupported	FILE, FTP, GOPHER, HTTP, HTTPS, NEWS, NNTP, WAIS				
RequestingUserName	TextSyntax				✓
Severity	ERROR, REPORT, WARNING				
SheetCollate	COLLATED, UNCOLLATED	✓	✓	✓	
Sides	ONE_SIDED, DUPLEX (или TWO_SIDED_, LONG_EDGE), TUMBLE (или TWO_SIDED_, SHORT_EDGE)	✓	✓	✓	

 **На заметку!** Как видите, многие из перечисленных в табл. 11.4 атрибутов являются узкоспециализированными и появились вследствие применения протокола Internet Printing Protocol 1.1 (по стандарту RFC 2911).

 **На заметку!** В более ранней версии прикладного программного интерфейса API для печати были внедрены классы **JobAttributes** и **PageAttributes**, которые служили той же цели, что и описываемые в этом разделе атрибуты печати. Но теперь эти классы уже считаются устаревшими.

`javax.print.attribute.Attribute 1.4`

- **Class getCategory()**
Получает категорию данного атрибута.
- **String getName()**
Получает имя данного атрибута.

`javax.print.attribute.AttributeSet 1.4`

- **boolean add(Attribute attr)**
Вводит атрибут в набор атрибутов. Если в наборе имеется другой атрибут из той же категории, он замещается новым атрибутом. Возвращает логическое значение **true**, если в результате выполнения данной операции набор атрибутов изменился.
- **Attribute get(Class category)**
Извлекает атрибут с ключом данной категории или пустое значение **null**, если такого атрибута не существует.
- **boolean remove(Attribute attr)**
- **boolean remove(Class category)**
Удаляют указанный атрибут или атрибут указанной категории из набора атрибутов. Возвращают логическое значение **true**, если в результате выполнения данной операции набор атрибутов изменился.
- **Attribute[] toArray()**
Возвращает массив со всеми атрибутами из данного набора.

`javax.print.PrintService 1.4`

- **PrintServiceAttributeSet getAttributes()**
Получает атрибуты данной службы печати.

`javax.print.DocPrintJob 1.4`

- **PrintJobAttributeSet getAttributes()**
Получает атрибуты текущего задания на печать.

На этом обсуждение средств печати изображений завершается. Теперь вы знаете, как выводить двухмерную графику и другие типы документов на печать, как перечислять печатающие устройства и потоковые службы печати и как устанавливать и извлекать атрибуты печати. Переядем далее к рассмотрению двух других средств, очень важных для создания пользовательских интерфейсов: буфера обмена и механизма перетаскивания объектов.

11.13. Буфер обмена

Одним из самых полезных и удобных в средах с ГПИ (вроде Windows и X Window System) является механизм *вырезания и вставки*. Он позволяет выбрать фрагмент данных в одной программе, вырезать или скопировать их в буфер обмена, а затем перейти в другую программу и вставить в ней содержимое буфера. Используя буфер обмена, можно переносить текст, рисунки или другие данные из одного документа в другой или из одной части документа в другую. Механизм вырезания и вставки настолько естествен, что большинство пользователей применяют его совершенно интуитивно.

Несмотря на то что действия, требующиеся для работы с буфером обмена, очень просты, реализация буфера намного сложнее, чем это может показаться на первый взгляд. Допустим, какой-нибудь фрагмент текста скопирован из текстового редактора в буфер обмена. При вставке этого текста в другой текстовый редактор было бы желательно, чтобы шрифты и прочие атрибуты форматирования данного фрагмента остались прежними. Это означает, что в буфере обмена должны также храниться сведения о форматировании. Но при вставке фрагмента текста в поле обычно предполагается получить только символы вставляемого текста без дополнительных кодов форматирования. Для обеспечения такой гибкости данных поставщик должен предоставить их в различных форматах, а потребитель — выбрать среди них подходящий формат.

Системный буфер обмена в Windows и Mac OS реализован практически одинаково. Эти реализации отличаются лишь в деталях. А в механизме работы с буфером обмена в X Window System на вырезание и вставку элементов с более сложным форматом, чем простой текст, накладываются некоторые ограничения. Эти ограничения следует учитывать, прорабатывая примеры программ, рассматриваемые в данном разделе.



На заметку! Для проверки типов объектов, которые могут передаваться между прикладными программами Java и системным буфером обмена, следует обратиться к файлу `jre/lib/flavormap.properties` на используемой платформе.

Нередко в прикладных программах требуется организовать копирование и вставку таких типов данных, которые не поддерживаются системным буфером обмена. В прикладном программном интерфейсе API для обмена данными поддерживаются ссылки на произвольные локальные объекты в рамках одной виртуальной машины. Между различными виртуальными машинами можно передавать сериализованные объекты и ссылки на удаленные объекты. В табл. 11.5 перечислены возможности механизма буфера обмена передавать данные.

Таблица 11.5. Возможности механизма буфера обмена передавать данные

Способ передачи данных	Формат данных
Между программой на Java и платформенно-ориентированной программой	Текст, рисунки, списки и т.п. (в зависимости от платформы)
Между двумя программами на Java	Сериализированные и удаленные объекты
В пределах одной программы на Java	Любые объекты

11.13.1. Классы и интерфейсы для передачи данных

Передача данных в Java реализована в пакете `java.awt.datatransfer`. Ниже перечислены некоторые особенности классов и интерфейсов из этого пакета.

- Объекты, передаваемые через буфер обмена, должны реализовывать интерфейс `Transferable`.
- Класс `Clipboard` описывает буфер обмена. Системный буфер обмена является конкретным примером реализации класса `Clipboard`.
- Класс `DataFlavor` описывает разновидности данных, которые могут быть размещены в буфере обмена.
- Чтобы получить уведомление о перезаписи содержимого буфера обмена, класс должен реализовать интерфейс `ClipboardOwner`. Право владеть буфером обмена позволяет выполнять "отложенное форматирование" сложных данных. Если программа передает простые данные (например, символьную строку), то она просто обновляет содержимое буфера обмена и переходит к выполнению следующих действий. Если же программа собирается вставить в буфер обмена сложные данные, которые могут быть по-разному отформатированы, то вряд ли стоит подготавливать все эти виды форматирования, так как многие из них могут вообще не понадобиться. Тем не менее данные нужно сохранить в буфере таким образом, чтобы впоследствии можно было обрабатывать типы документов по мере надобности. Поэтому владелец буфера обмена с помощью метода `lostOwnership()` должен быть своевременно уведомлен об изменении его содержимого. Изменение содержимого буфера обмена означает, что сведения о сложных данных больше не требуются. В приведенных далее примерах механизм, управляющий правами владения буфером обмена, не применяется.

11.13.2. Передача текста

Изучение классов передачи данных лучше всего начать с простейшей ситуации: передачи текста через системный буфер обмена. Сначала следует получить ссылку на системный буфер обмена:

```
Clipboard clipboard =
    Toolkit.getDefaultToolkit().getSystemClipboard();
```

Строки, которые передаются в буфер обмена, необходимо заключить в оболочку объекта типа `StringSelection` с помощью следующего конструктора:

```
String text = . . .
StringSelection selection = new StringSelection(text);
```

Передача данных, по существу, происходит при вызове метода `setContents()`, который принимает объекты типа `StringSelection` и `ClipBoardOwner` в качестве параметров. Если владелец буфера обмена не имеет значения, то второму параметру данного метода следует присвоить пустое значение `null`:

```
clipboard.setContents(selection, null);
```

Ниже показано, каким образом выполняется обратная операция чтения строки из буфера обмена.

```
DataFlavor flavor = DataFlavor.stringFlavor;
if (clipboard.isDataFlavorAvailable(flavor))
    String text = (String) clipboard.getData(flavor);
```

В листинге 11.17 приведен исходный код примера программы, демонстрирующей операции копирования и вставки текста в приложении Java через системный буфер обмена. Если выбрать фрагмент текста в текстовой области и щелкнуть на кнопке `Copy` в этой программе, выбранный текст копируется в системный буфер обмена, после чего его можно вставить в окно любого текстового редактора (рис. 11.40). Имеется также возможность выполнить обратную операцию, скопировав текст из окна любого текстового редактора и вставив его в демонстрационную программу, для чего достаточно щелкнуть в ней на кнопке `Paste`.

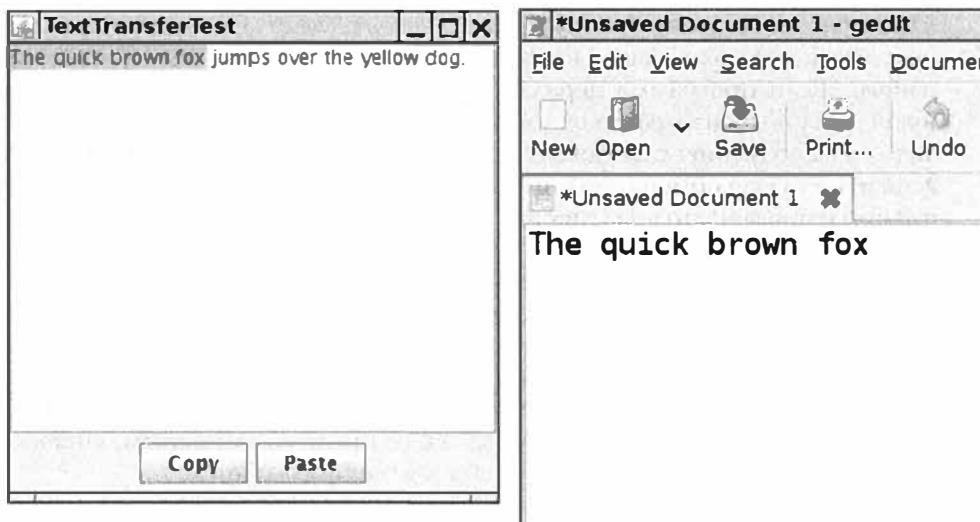


Рис. 11.40. Программа `TextTransferTest`

Листинг 11.17. Исходный код программы из файла `transferText/TextTransferFrame.java`

```
1 package transferText;
2
3 import java.awt.*;
4 import java.awt.datatransfer.*;
5 import java.awt.event.*;
6 import java.io.*;
```

```
7 import javax.swing.*;
8
9 /**
10  * Этот фрейм содержит текстовую область и кнопки
11  * для копирования и вставки текста
12 */
13 public class TextTransferFrame extends JFrame
14 {
15     private JTextArea textArea;
16     private static final int TEXT_ROWS = 20;
17     private static final int TEXT_COLUMNS = 60;
18
19     public TextTransferFrame()
20     {
21         textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
22         add(new JScrollPane(textArea), BorderLayout.CENTER);
23         JPanel panel = new JPanel();
24
25         JButton copyButton = new JButton("Copy");
26         panel.add(copyButton);
27         copyButton.addActionListener(event -> copy());
28
29         JButton pasteButton = new JButton("Paste");
30         panel.add(pasteButton);
31         pasteButton.addActionListener(event -> paste());
32
33         add(panel, BorderLayout.SOUTH);
34         pack();
35     }
36
37 /**
38  * Копирует выбранный текст в системный буфер обмена
39  */
40 private void copy()
41 {
42     Clipboard clipboard =
43         Toolkit.getDefaultToolkit().getSystemClipboard();
44     String text = textArea.getSelectedText();
45     if (text == null) text = textArea.getText();
46     StringSelection selection = new StringSelection(text);
47     clipboard.setContents(selection, null);
48 }
49
50 /**
51  * Вставляет текст из системного буфера обмена
52  * в текстовую область
53  */
54 private void paste()
55 {
56     Clipboard clipboard =
57         Toolkit.getDefaultToolkit().getSystemClipboard();
58     DataFlavor flavor = DataFlavor.stringFlavor;
59     if (clipboard.isDataFlavorAvailable(flavor))
60     {
61         try
62         {
63             String text = (String) clipboard.getData(flavor);
64             textArea.replaceSelection(text);
```

```
65      }
66      catch (UnsupportedFlavorException e | IOException ex)
67      {
68          JOptionPane.showMessageDialog(this, ex);
69      }
70  }
71 }
72 }
```

java.awt.Toolkit 1.0

- **Clipboard getSystemClipboard() 1.1**
Получает содержимое системного буфера обмена.

java.awt.datatransfer.Clipboard 1.1

- **Transferable getContents(Object requester)**

Получает содержимое буфера обмена.

Параметры: **requester**

Объект, который
запрашивает содержимое
буфера обмена. Значение
этого параметра на самом
деле не используется

- **void setContents(Transferable contents, ClipboardOwner owner)**

Размещает содержимое в буфере обмена.

Параметры: **contents**

Объект, инкапсулирующий
содержимое буфера обмена
owner
Объект, который должен
быть уведомлен об
изменении содержимого
буфера обмена (методом
lostOwnership()), а если
уведомление не требуется, то
указывается пустое значение
null данного параметра

- **boolean isDataFlavorAvailable(DataFlavor flavor) 5.0**

Возвращает логическое значение **true**, если в буфере обмена содержатся данные указанной разновидности.

- **Object getData(DataFlavor flavor) 5.0**

Получает данные указанной разновидности, а если они недоступны, то генерирует исключение типа **UnsupportedFlavorException**.

java.awt.datatransfer.ClipboardOwner 1.1

- **void lostOwnership(Clipboard clipboard, Transferable contents)**
Уведомляет объект, что он больше не является владельцем содержимого буфера обмена.

Параметры:	clipboard	Буфер обмена, содержимое которого изменилось
	contents	Элемент, размещаемый в буфере обмена данным владельцем

java.awt.datatransfer.Transferable 1.1

- **boolean isDataFlavorSupported(DataFlavor flavor)**
Возвращает логическое значение **true**, если поддерживается указанная разновидность данных, в противном случае — логическое значение **false**.
- **Object getTransferData(DataFlavor flavor)**
Возвращает данные, отформатированные по указанной разновидности. Генерирует исключение типа **UnsupportedFlavorException**, если указанная разновидность данных не поддерживается.

11.13.3. Интерфейс **Transferable** и разновидности данных

Объект типа **DataFlavor**, представляющий разновидность данных, обладает следующими характеристиками.

- Тип MIME (например, `image/gif`).
- Класс представления, предназначенный для доступа к данным (например, `java.awt.Image`).

Кроме того, каждая разновидность данных имеет свое удобочитаемое название (например, "GIF Image"). Класс представления может быть указан с помощью параметра **class** в типе MIME следующим образом:

`image/gif;class=java.awt.Image`



На заметку! Это всего лишь пример, который иллюстрирует синтаксис указания разновидности данных. Никакой стандартной разновидности для передачи графических данных изображения в формате GIF не существует.

Если параметр **class** не задан, то в качестве класса представления используется поток ввода типа **InputStream**. Для передачи локальных, сериализованных и удаленных объектов в Java предусмотрены следующие три типа MIME:

`application/x-java-jvm-local-objectref`
`application/x-java-serialized-object`
`application/x-java-remote-object`



На заметку! Префикс **x-** означает экспериментальное имя, которое еще не утверждено IANA (Комитет по цифровым адресам в Интернете) в качестве стандартного типа MIME.

Например, стандартная разновидность данных `StringFlavor` описывается с помощью следующего типа MIME:

```
application/x-java-serialized-object;class=java.lang.String
```

У буфера обмена можно также запрашивать список всех доступных разновидностей данных следующим образом:

```
DataFlavor[] flavors = clipboard.getAvailableDataFlavors()
```

Кроме того, для буфера обмена можно устанавливать приемник событий типа `FlavorListener`, который будет уведомляться об изменении любой разновидности данных в буфере обмена. Подробнее об этом приемнике событий см. ниже в описании соответствующего прикладного программного интерфейса API.

`java.awt.datatransfer.DataFlavor 1.1`

- **`DataFlavor(String mimeType, String humanPresentableName)`**

Создает разновидность данных, описывающую потоковые данные в формате, соответствующем указанному типу MIME.

Параметры: `mimeType`

Символьная строка,
представляющая тип MIME

`humanPresentableName`

Удобочитаемое имя
разновидности данных

- **`DataFlavor(Class class, String humanPresentableName)`**

Создает разновидность данных, описывающую класс платформы Java со следующим типом MIME: `applications/x-java-serialized-object;class=имяКласса`.

Параметры: `class`

Класс, извлекаемый с
помощью объекта типа

`Transferable`

`umanPresentableName`

Удобочитаемое имя
разновидности данных

- **`String getMimeType()`**

Возвращает строку, представляющую тип MIME.

- **`boolean isMimeTypeEqual(String mimeType)`**

Проверяет соответствие разновидности данных указанному типу MIME.

- **`String getHumanPresentableName()`**

Возвращает удобочитаемое имя формата для разновидности данных.

- **`Class getRepresentationClass()`**

Возвращает объект типа `Class`, представляющий класс объекта, который возвращается объектом типа `Transferable` для разновидности данных. Это может быть параметр `class` типа MIME или поток ввода типа `InputStream`.

java.awt.datatransfer.Clipboard 1.1

- **DataFlavor[] getAvailableDataFlavors() 5.0**
Возвращает массив доступных разновидностей данных.
- **void addFlavorListener(FlavorListener listener) 5.0**

Добавляет приемник событий, который уведомляется об изменениях в наборе доступных разновидностей данных.

java.awt.datatransfer.Transferable 1.1

- **DataFlavor[] getTransferDataFlavors()**
Возвращает массив поддерживаемых разновидностей данных.

java.awt.datatransfer.FlavorListener 5.0

- **void flavorsChanged(FlavorEvent event)**
Вызывается при изменениях в наборе доступных разновидностей данных.

11.13.4. Передача изображений через буфер обмена

Объекты, которые требуется передать через буфер обмена, должны реализовывать интерфейс Transferable. В настоящее время этот интерфейс реализует в стандартной библиотеке Java единственный открытый класс StringSelection. В этом разделе показано, каким образом изображения передаются через буфер обмена. В стандартной библиотеке Java не предусмотрен класс для такой передачи, поэтому его придется реализовать самостоятельно.

Создать такой класс совсем не трудно. Он просто сообщает, что единственной доступной разновидностью данных является DataFlavor.imageFlavor и что она содержит объект image для передачи изображения:

```
class ImageTransferable implements Transferable
{
    private Image theImage;

    public ImageTransferable(Image image)
    {
        theImage = image;
    }

    public DataFlavor[] getTransferDataFlavors()
    {
        return new DataFlavor[] { DataFlavor.imageFlavor };
    }

    public boolean isDataFlavorSupported(DataFlavor flavor)
    {
        return flavor.equals(DataFlavor.imageFlavor);
    }
}
```

```

public Object getTransferData(DataFlavor flavor)
    throws UnsupportedFlavorException
{
    if(flavor.equals(DataFlavor.imageFlavor))
    {
        return theImage;
    }
    else
    {
        throw new UnsupportedFlavorException(flavor);
    }
}
}

```



На заметку! В Java SE предоставляется константа `DataFlavor.imageFlavor` и автоматически выполняются все необходимые действия по преобразованию изображений, сформированных средствами Java, в изображения собственного формата буфера обмена. Но, как ни странно, в комплекте JDK не предусмотрен класс-оболочка, чтобы разместить изображение в буфере обмена.

В примере программы, исходный код которой приведен в листинге 11.18, демонстрируется передача изображений из приложения Java в другое приложение и обратно через системный буфер обмена. После запуска на выполнение эта программа формирует изображение, состоящее из красного круга. Щелкните на кнопке *Copy*, чтобы скопировать это изображение в буфер обмена, а затем вставьте его в другое приложение (рис. 11.41). Скопируйте изображение из другого приложения в системный буфер обмена и щелкните на кнопке *Paste*, чтобы вставить его обратно в исходную программу (рис. 11.42).

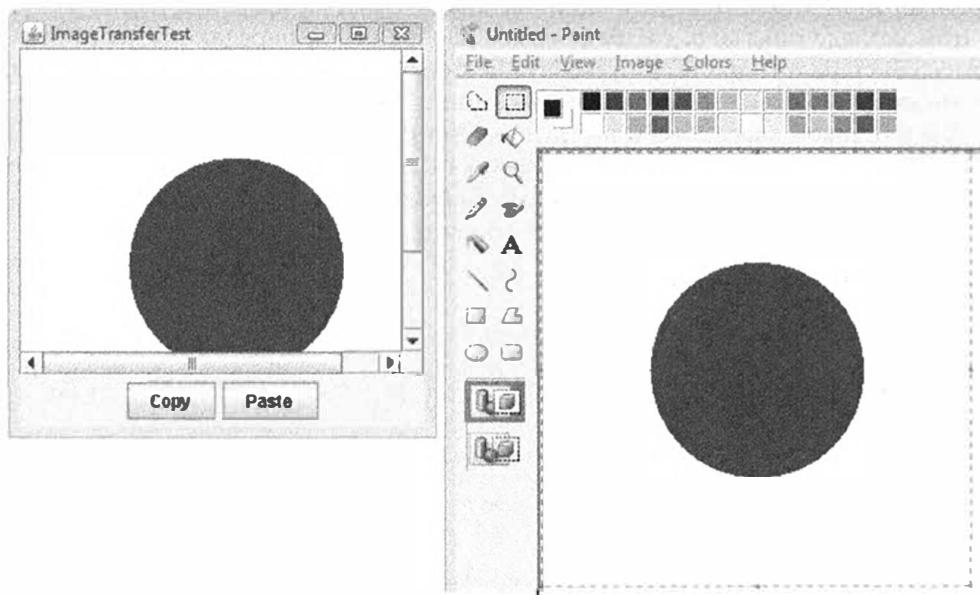


Рис. 11.41. Копирование изображения из прикладной программы на Java в платформенно-ориентированную программу

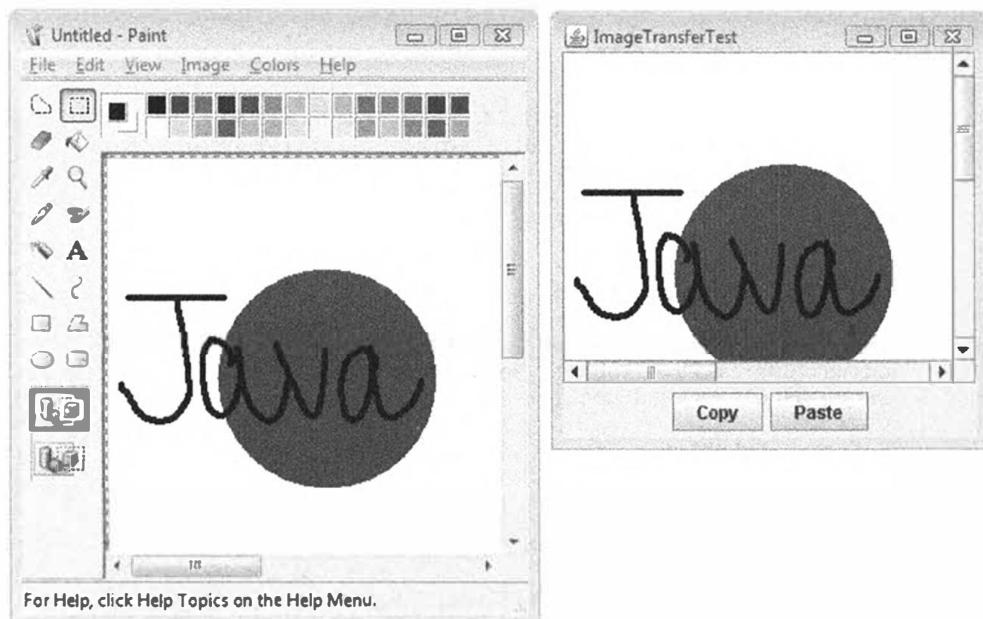


Рис. 11.42. Копирование изображения из платформенно-ориентированной программы в прикладную программу на Java

Данная программа является усовершенствованной версией программы из предыдущего примера, где демонстрировалась передача текста через буфер обмена. В данном же примере используется разновидность данных `DataFlavor.imageFlavor`, а для передачи изображения через системный буфер обмена — класс `ImageTransferable`.

Листинг 11.18. Исходный код из файла `imageTransfer/ImageTransferFrame.java`

```

1 package imageTransfer;
2
3 import java.awt.*;
4 import java.awt.datatransfer.*;
5 import java.awt.image.*;
6 import java.io.*;
7
8 import javax.swing.*;
9
10 /**
11  * Этот фрейм содержит метку изображения и кнопки
12  * для копирования и вставки изображения
13  */
14 class ImageTransferFrame extends JFrame
15 {
16     private JLabel label;
17     private Image image;
18     private static final int IMAGE_WIDTH = 300;
19     private static final int IMAGE_HEIGHT = 300;

```

```
20
21 public ImageTransferFrame()
22 {
23     label = new JLabel();
24     image = new BufferedImage(IMAGE_WIDTH, IMAGE_HEIGHT,
25                             BufferedImage.TYPE_INT_ARGB);
26     Graphics g = image.getGraphics();
27     g.setColor(Color.WHITE);
28     g.fillRect(0, 0, IMAGE_WIDTH, IMAGE_HEIGHT);
29     g.setColor(Color.RED);
30     g.fillOval(IMAGE_WIDTH / 4, IMAGE_WIDTH / 4,
31                 IMAGE_WIDTH / 2, IMAGE_HEIGHT / 2);
32
33     label.setIcon(new ImageIcon(image));
34     add(new JScrollPane(label), BorderLayout.CENTER);
35     JPanel panel = new JPanel();
36
37     JButton copyButton = new JButton("Copy");
38     panel.add(copyButton);
39     copyButton.addActionListener(event -> copy());
40
41     JButton pasteButton = new JButton("Paste");
42     panel.add(pasteButton);
43     pasteButton.addActionListener(event -> paste());
44
45     add(panel, BorderLayout.SOUTH);
46     pack();
47 }
48
49 /**
50 * Копирует текущее изображение в системный буфер обмена
51 */
52 private void copy()
53 {
54     Clipboard clipboard =
55         Toolkit.getDefaultToolkit().getSystemClipboard();
56     ImageTransferable selection = new ImageTransferable(image);
57     clipboard.setContents(selection, null);
58 }
59
60 /**
61 * Вставляет изображение из системного буфера
62 * на место метки изображения
63 */
64 private void paste()
65 {
66     Clipboard clipboard =
67         Toolkit.getDefaultToolkit().getSystemClipboard();
68     DataFlavor flavor = DataFlavor.imageFlavor;
69     if (clipboard.isDataFlavorAvailable(flavor))
70     {
71         try
72         {
73             image = (Image) clipboard.getData(flavor);
74             label.setIcon(new ImageIcon(image));
75         }
76         catch (UnsupportedFlavorException | IOException ex)
77         {
```

```

78         JOptionPane.showMessageDialog(this, ex);
79     }
80 }
81 }
82 }

```

11.13.5. Передача объектов Java через системный буфер обмена

Допустим, объекты требуется копировать и вставлять из одной прикладной программы на Java в другую. Для этого достаточно разместить в системном буфере обмена объекты, сериализованные средствами Java.

Такая возможность демонстрируется в примере программы из листинга 11.19. Данная программа отображает окно селектора цвета. Если щелкнуть на кнопке Copy, она скопирует в системный буфер обмена выбранный цвет в виде сериализованного объекта типа Color. А если щелкнуть на кнопке Paste, то данная программа проверит, содержит ли в системном буфере объект типа Color, и тогда он извлекается оттуда и становится текущим выбранным цветом.

Сериализованные объекты можно также передавать из одной прикладной программы на Java в другую (рис. 11.43). Чтобы убедиться в этом, запустите два экземпляра программы SerialTransferTest, затем щелкните на кнопке Copy в первом ее экземпляре и на кнопке Paste во втором. В итоге объект типа Color будет передан из одной виртуальной машины Java в другую.

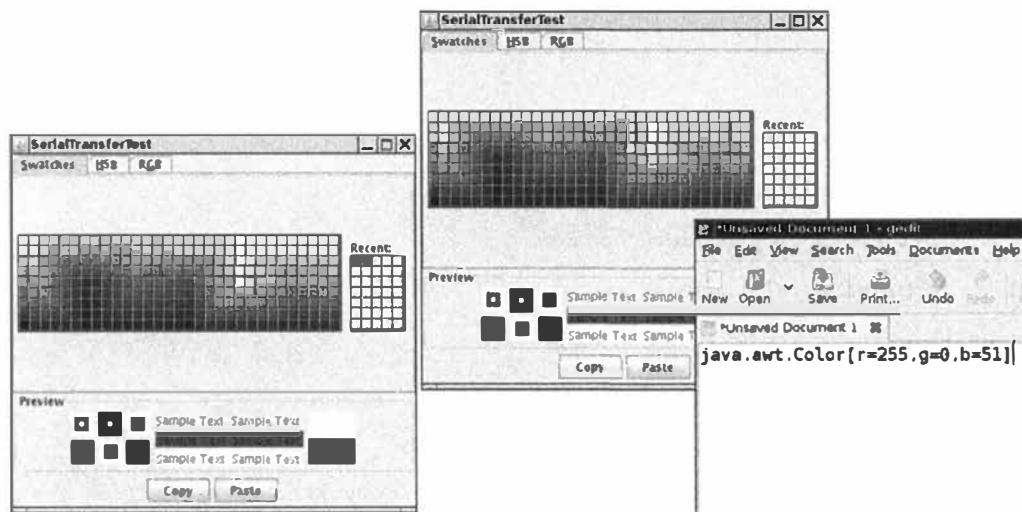


Рис. 11.43. Копирование данных между двумя экземплярами прикладной программы на Java

Для передачи данных на платформе Java предусмотрено размещение двоичных данных сериализованного объекта в системном буфере, чтобы другая программа на Java (не обязательно такого же типа, как та, что сформировала данные для буфера обмена) смогла извлечь данные из буфера и десериализовать объект. Разумеется, программа, написанная не на Java, вряд ли будет знать, что делать с данными в буфере обмена. Именно поэтому в рассматриваемом здесь

примере программы данные из буфера обмена предоставляются в другой разновидности: в текстовом виде. Текст получается в результате выполнения метода `toString()` для передаваемого объекта. Чтобы увидеть текст, запустите данную программу на выполнение, выберите какой-нибудь цвет и щелкните на кнопке `Paste` в текстовом редакторе. Это приведет к появлению в окне текстового редактора строки, аналогичной следующей:

```
java.awt.Color[r=255,g=0,b=51]
```

По существу, никакого дополнительного кода для передачи сериализуемых объектов не требуется. Для этой цели достаточно воспользоваться типом MIME следующим образом:

```
application/x-java-serialized-object;class=имяКласса
```

Класс-оболочку для передачи данных, как и прежде, придется создавать самостоятельно. В исходном коде из листинга 11.19 показано, как это делается.

Листинг 11.19. Исходный код из файла serialTransfer/SerialTransferFrame.java

```
1 package serialTransfer;
2
3 import java.awt.*;
4 import java.awt.datatransfer.*;
5 import java.awt.event.*;
6 import java.io.*;
7 import javax.swing.*;
8
9 /**
10  * Этот фрейм содержит селектор цвета и кнопки
11  * для копирования и вставки
12  */
13 class SerialTransferFrame extends JFrame
14 {
15     private JColorChooser chooser;
16
17     public SerialTransferFrame()
18     {
19         chooser = new JColorChooser();
20         add(chooser, BorderLayout.CENTER);
21         JPanel panel = new JPanel();
22
23         JButton copyButton = new JButton("Copy");
24         panel.add(copyButton);
25         copyButton.addActionListener(event -> copy());
26
27         JButton pasteButton = new JButton("Paste");
28         panel.add(pasteButton);
29         pasteButton.addActionListener(event -> paste());
30
31         add(panel, BorderLayout.SOUTH);
32         pack();
33     }
34
35 /**
36  * Вставляет цвет из системного буфера обмена в селектор цвета
37  */
```

```
38     private void copy()
39     {
40         Clipboard clipboard =
41             Toolkit.getDefaultToolkit().getSystemClipboard();
42         Color color = chooser.getColor();
43         Serializable selection =
44             new Serializable(color);
45         clipboard.setContents(selection, null);
46     }
47
48 /**
49 * Этот класс служит оболочкой для передачи данных
50 * в виде сериализуемых объектов
51 */
52 private void paste()
53 {
54     Clipboard clipboard =
55         Toolkit.getDefaultToolkit().getSystemClipboard();
56     try
57     {
58         DataFlavor flavor = new DataFlavor(
59             "application/x-java-serialized-object",
60             "class=java.awt.Color");
61         if (clipboard.isDataFlavorAvailable(flavor))
62         {
63             Color color = (Color) clipboard.getData(flavor);
64             chooser.setColor(color);
65         }
66     }
67     catch (ClassNotFoundException
68          | UnsupportedFlavorException | IOException ex)
69     {
70         JOptionPane.showMessageDialog(this, ex);
71     }
72 }
73 }
74
75 /**
76 * Этот класс служит оболочкой для передачи данных
77 * в виде сериализуемых объектов
78 */
79 class Serializable implements Transferable
80 {
81     private Serializable obj;
82
83 /**
84 * Конструирует выбираемый объект
85 * @param o Любой сериализуемый объект
86 */
87 Serializable(Serializable o)
88 {
89     obj = o;
90 }
91
92 public DataFlavor[] getTransferDataFlavors()
93 {
94     DataFlavor[] flavors = new DataFlavor[2];
95     Class<?> type = obj.getClass();
```

```

96     String mimeType = "application/x-java-serialized-object";
97             class=" + type.getName();
98     try
99     {
100         flavors[0] = new DataFlavor(mimeType);
101         flavors[1] = DataFlavor.stringFlavor;
102         return flavors;
103     }
104     catch (ClassNotFoundException e)
105     {
106         return new DataFlavor[0];
107     }
108 }
109
110 public boolean isDataFlavorSupported(DataFlavor flavor)
111 {
112     return DataFlavor.stringFlavor.equals(flavor)
113         || "application".equals(flavor.getPrimaryType())
114         && "x-java-serialized-object"
115             .equals(flavor.getSubType())
116         && flavor.getRepresentationClass()
117             .isAssignableFrom(obj.getClass());
118 }
119
120 public Object getTransferData(DataFlavor flavor)
121     throws UnsupportedFlavorException
122 {
123     if (!isDataFlavorSupported(flavor))
124         throw new UnsupportedFlavorException(flavor);
125
126     if (DataFlavor.stringFlavor.equals(flavor))
127         return obj.toString();
128
129     return obj;
130 }
131 }
```

11.13.6. Передача ссылок на объекты через локальный буфер обмена

Иногда требуется копировать и вставлять данных таких типов, которые не поддерживаются системным буфером обмена, т.е. не являются сериализуемыми. Для передачи ссылки на произвольный объект в пределах одной и той же виртуальной машины Java (JVM) применяется тип MIME:

`application/x-java-jvm-local-objectref;class=имяКласса`

Для этого требуется определить объект-оболочку типа `Transferable`. Этот процесс аналогичен процессу определения объекта-оболочки типа `SerialTransferable`, рассмотренному в предыдущем разделе. Ссылка на объект имеет смысл только в пределах одной виртуальной машины. Поэтому копировать такой объект в системный буфер обмена нельзя. Вместо этого следует использовать локальный буфер обмена, как показано ниже. В качестве параметра конструктору приведенного ниже класса следует передать имя буфера обмена.

```
Clipboard clipboard = new Clipboard("local");
```

Но у передачи данных через локальный буфер обмена имеется один серьезный недостаток. Локальный и системный буфера обмена требуется синхронизировать, чтобы пользователи не путали их. В настоящий момент подобная синхронизация на платформе Java пока еще не поддерживается.

java.awt.datatransfer.Clipboard 1.1

- **Clipboard(String name)**

Создает локальный буфер обмена с указанным именем.

11.14. Перетаскивание объектов

При использовании механизма копирования и вставки для передачи данных между двумя программами в роли посредника всегда выступает буфер обмена. Механизм *перетаскивания* исключает участие такого посредника и позволяет двум программам взаимодействовать напрямую. На платформе Java предусмотрена элементарная поддержка данного механизма. В частности, она позволяет выполнять операции перетаскивания между программами на Java и платформенно-ориентированными программами. В этом разделе приведен пример создания прикладной программы на Java для выполнения функций как источника, так и приемника перетаскивания.

Прежде чем углубляться в изучение поддержки операций перетаскивания, предоставляемой на платформе Java, рассмотрим вкратце пользовательский интерфейс перетаскивания на примере стандартных приложений Windows Explorer и WordPad. На другой платформе вместо этих приложений можно выбрать для экспериментирования другие стандартные прикладные программы с возможностями перетаскивания.

Операция *перетаскивания* инициируется выполнением соответствующего жеста в источнике перетаскивания, т.е. сначала выбирается один объект или более, а затем они перетаскиваются из исходного места в какое-нибудь другое. После отпускания кнопки мыши над местом назначения, т.е. приемником перетаскивания, последний запрашивает у источника сведения о перетаскиваемых объектах и выполняет соответствующую операцию. Так, если перетащить пиктограмму файла на пиктограмму какого-нибудь каталога в окне диспетчера файлов, этот файл переместится в выбранный каталог. Но если перетащить пиктограмму файла в текстовый редактор, то этот файл откроется в окне текстового редактора. (Очевидно, что в данном случае подразумевается использование такого диспетчера файлов или текстового редактора, который поддерживает операции перетаскивания, как, например, стандартные приложения Explorer и WordPad в Windows или Nautilus и gedit в Gnome.)

Если при перетаскивании файла в Windows нажать дополнительно клавишу <Ctrl>, вместо операции *перемещения* будет выполнена операция *копирования* в каталог только копии файла. А если одновременно нажать клавиши <Ctrl> и <Shift>, то в каталоге будет создан ярлык в качестве ссылки на данный файл. (На других

платформах для выполнения этих операций могут быть предусмотрены другие комбинации клавиш.)

Таким образом, операции перетаскивания могут подразумевать выполнение трех различных действий:

- перемещение;
- копирование;
- создание ярлыка.

Под созданием ярлыка, по существу, подразумевается создание ссылки на перетаскиваемый объект. Такие ссылки обычно требуют поддержки со стороны обслуживающей операционной системы (вроде символьических ссылок на файлы или объектных ссылок на компоненты документа) и обычно не имеют особого смысла в межплатформенных программах. Поэтому в этом разделе основное внимание уделяется применению только тех операций перетаскивания, которые подразумевают копирование и перемещение объектов.

Обычно операции перетаскивания сопровождаются специальными визуальными эффектами. Самым простым из них является изменение вида курсора мыши. При наведении курсора на потенциальный приемник перетаскивания его вид изменяется, указывая на возможность выполнения операции перетаскивания. Если перетаскивание возможно, то вид курсора дополнительно указывает, какое именно действие разрешается выполнить. В табл. 11.6 представлены некоторые виды курсора, которые он может принимать при перетаскивании.

Таблица 11.6. Виды курсора, которые он может принимать при перетаскивании

Действие	Значок в Windows	Значок в Gnome
Перемещение		
Копирование		
Создание ярлыка		
Перетаскивание запрещено		

Перетаскивать можно не только пиктограммы файлов, но и другие объекты. Например, можно выделить фрагмент текста в редакторе WordPad или gedit и перетащить его. Попробуйте перетащить фрагменты текста в разные места назначения и посмотрите, что при этом произойдет.



На заметку! Подобный эксперимент демонстрирует недостатки механизма перетаскивания как части пользовательского интерфейса. Он может вызывать у пользователей следующие вопросы: какие объекты и куда можно перетаскивать и к чему это приведет. Вследствие того что операция перемещения, выполняемая по умолчанию при перетаскивании, может приводить к удалению оригинала, многие пользователи, по вполне понятным причинам, стараются не особенно экспериментировать с перетаскиванием.

11.14.1. Поддержка передачи данных в Swing

В версии Java SE 1.4 в ряд компонентов Swing была встроена поддержка операций перетаскивания (табл. 11.7). Одни из этих компонентов позволяют выбирать текст для перетаскивания, а другие — вставлять его. Для обеспечения обратной совместимости следует вызвать метод `setDragEnabled()`, активизирующий перетаскивание, тогда как отпускание активно всегда.

Таблица 11.7. Поддержка передачи данных в компонентах Swing

Компонент	Источник перетаскивания	Приемник перетаскивания
<code>JFileChooser</code>	Экспортирует список файлов	Отсутствует
<code>JColorChooser</code>	Экспортирует объект, представляющий цвет	Принимает объекты, представляющие разные цвета
<code>JTextField</code>	Экспортирует выбранный текст	Принимает текст
<code>JFormattedTextField</code>		
<code>JPasswordField</code>	Отсутствует (из соображений безопасности)	Принимает текст
<code>JTextArea</code>	Экспортирует выбранный текст	Принимает текст и списки файлов
<code>JTextPane</code>		
<code>JEditorPane</code>		
<code>JList</code>	Экспортирует описание выбранного текста (т.е. только его копию)	Отсутствует
<code>JTable</code>		
<code>JTree</code>		



На заметку! В состав пакета `java.awt.dnd` входит низкоуровневый прикладной программный интерфейс API для перетаскивания. Он составляет основу для поддержки перетаскивания в библиотеке Swing. Этот прикладной интерфейс в данной книге не рассматривается.

В примере программы, исходный код которой приведен в листинге 11.20, демонстрируется поведение всех этих компонентов. После запуска этой программы на выполнение обратите внимание на ряд моментов.

- Из списка, таблицы и дерева можно выбирать и перетаскивать несколько элементов, как следует из листинга 11.21.
- Перетаскивать элементы из таблицы не совсем удобно: сначала нужно выбрать элемент мышью, затем отпустить кнопку мыши и снова щелкнуть ею и только затем выполнить перетаскивание.
- При перетаскивании элементов в текстовую область можно заметить, как происходит форматирование перетаскиваемых данных. В частности, ячейки таблицы разделяются символами табуляции, а каждая выбранная строка таблицы размещается в отдельной строке (рис. 11.44).
- Из списка, таблицы, селектора файлов и селектора цветов элементы можно только копировать, но не перемещать. Удаление элементов из списка, таблицы и дерева не допускается ни в одной из моделей данных. О том, как реализовать возможность редактирования в модели данных, речь пойдет в следующем разделе.

- Перетаскивать элементы в список, таблицу, дерево или селектор файлов нельзя.
- Если запустить на выполнение две копии программы, то образец цвета можно перетаскивать из одного селектора цвета в другой.
- Перетаскивать текст из текстовой области нельзя, поскольку для нее не вызывается метод `setDragEnabled()`.

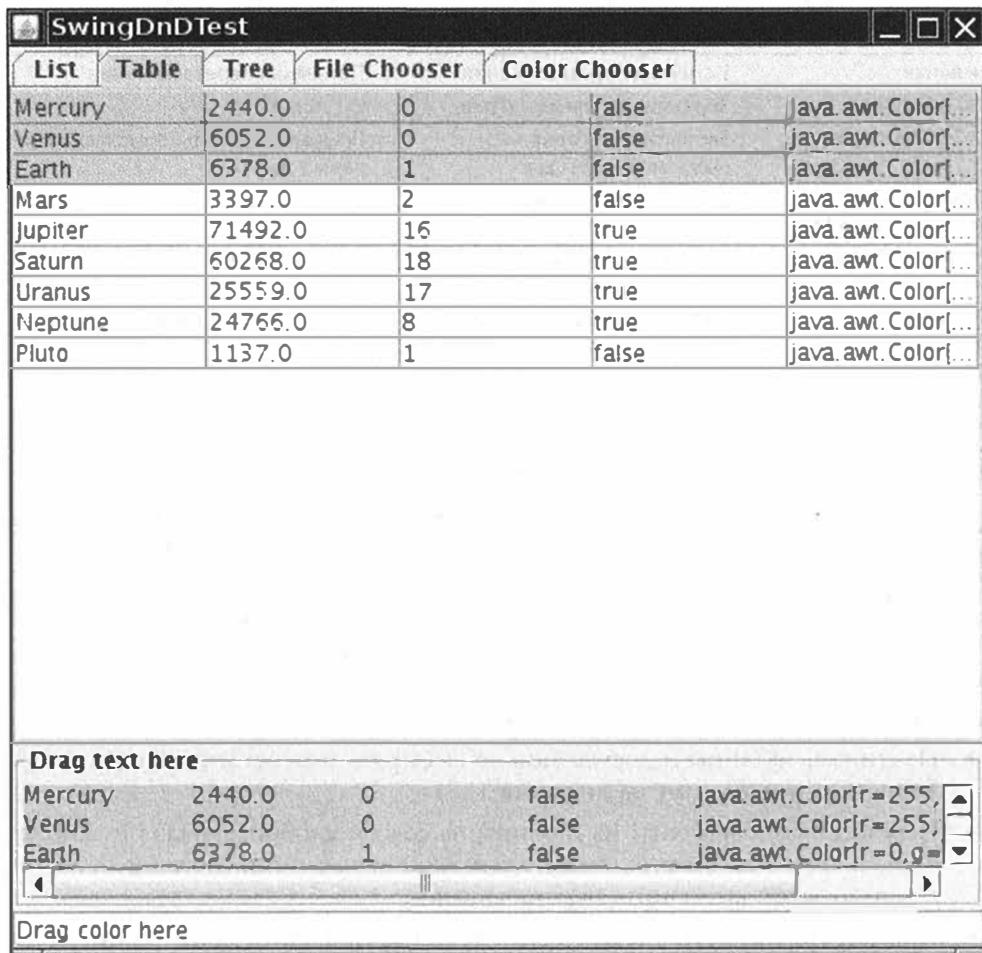


Рис. 11.44. Программа для тестирования поддержки перетаскивания в библиотеке Swing

В библиотеке Swing предусмотрен потенциально полезный механизм для быстрого превращения компонента из источника перетаскивания в приемник. С этой целью можно установить обработчик передачи данных для заданного свойства. Так, в рассматриваемом здесь примере программы делается следующий вызов:

```
textField.setTransferHandler(new TransferHandler("background"));
```

Благодаря этому перетаскивание цвета в текстовое поле будет приводить к изменению цвета его фона. При перетаскивании этот обработчик будет проверять, имеется ли у одной из разновидностей данных класс представления Color. И если таковой имеется, то вызывается метод setBackground().

Установка упомянутого выше обработчика передачи данных в текстовом поле означает отключение стандартного обработчика передачи данных, из-за чего в этом текстовом поле больше нельзя будет ни вырезать, ни вставлять, ни перетаскивать текст, но в то же время из него можно будет перетаскивать цвет. Чтобы инициировать жест перетаскивания, придется выбрать какой-нибудь фрагмент текста. При перетаскивании этого текста обнаружится, что его можно опустить в селекторе цвета, изменив цвет в его образце на цвет фона текстового поля. Но перетащить текст в саму текстовую область не удастся.

Листинг 11.20. Исходный код из файла dnd/SwingDnDTest.java

```
1 package dnd;
2
3 import java.awt.*;
4 import javax.swing.*;
5
6 /**
7  * В этой программе демонстрируется элементарная
8  * поддержка операций перетаскивания и опускания
9  * объектов в библиотеке Swing
10 * @version 1.11 2016-05-10
11 * @author Cay Horstmann
12 */
13 public class SwingDnDTest
14 {
15     public static void main(String[] args)
16     {
17         EventQueue.invokeLater(() ->
18         {
19             JFrame frame = new SwingDnDFrame();
20             frame.setTitle("SwingDnDTest");
21             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22             frame.setVisible(true);
23         });
24     }
25 }
```

Листинг 11.21. Исходный код из файла dnd/SampleComponents.java

```
1 package dnd;
2
3 import java.awt.*;
4
5 import javax.swing.*;
6 import javax.swing.tree.*;
7
8 public class SampleComponents
9 {
10     public static JTree tree()
```

```
11  {
12      DefaultMutableTreeNode root =
13          new DefaultMutableTreeNode("World");
14      DefaultMutableTreeNode country =
15          new DefaultMutableTreeNode("USA");
16      root.add(country);
17      DefaultMutableTreeNode state =
18          new DefaultMutableTreeNode("California");
19      country.add(state);
20      DefaultMutableTreeNode city =
21          new DefaultMutableTreeNode("San Jose");
22      state.add(city);
23      city = new DefaultMutableTreeNode("Cupertino");
24      state.add(city);
25      state = new DefaultMutableTreeNode("Michigan");
26      country.add(state);
27      city = new DefaultMutableTreeNode("Ann Arbor");
28      state.add(city);
29      country = new DefaultMutableTreeNode("Germany");
30      root.add(country);
31      state = new DefaultMutableTreeNode("Schleswig-Holstein");
32      country.add(state);
33      city = new DefaultMutableTreeNode("Kiel");
34      state.add(city);
35      return new JTree(root);
36  }
37
38  public static JList<String> list()
39  {
40      String[] words = { "quick", "brown", "hungry", "wild",
41                         "silent", "huge", "private", "abstract",
42                         "static", "final" };
43
44      DefaultListModel<String> model = new DefaultListModel<>();
45      for (String word : words)
46          model.addElement(word);
47      return new JList<>(model);
48  }
49
50  public static JTable table()
51  {
52      Object[][] cells = {
53          { "Mercury", 2440.0, 0, false, Color.YELLOW },
54          { "Venus", 6052.0, 0, false, Color.YELLOW },
55          { "Earth", 6378.0, 1, false, Color.BLUE },
56          { "Mars", 3397.0, 2, false, Color.RED },
57          { "Jupiter", 71492.0, 16, true, Color.ORANGE },
58          { "Saturn", 60268.0, 18, true, Color.ORANGE },
59          { "Uranus", 25559.0, 17, true, Color.BLUE },
60          { "Neptune", 24766.0, 8, true, Color.BLUE },
61          { "Pluto", 1137.0, 1, false, Color.BLACK } };
62
63      String[] columnNames = { "Planet", "Radius",
64                             "Moons", "Gaseous", "Color" };
65      return new JTable(cells, columnNames);
66  }
67 }
```

```
javax.swing.JComponent 1.2
```

- **void setTransferHandler(TransferHandler handler) 1.4**

Устанавливает обработчик для управления операциями передачи данных (т.е. операциями вырезания, копирования, вставки и перетаскивания).

```
javax.swing.TransferHandler 1.4
```

- **TransferHandler(String propertyName)**

Создает обработчик передачи данных для чтения или записи свойства компонента JavaBeans с указанным именем при выполнении операции передачи данных.

```
javax.swing.JFileChooser 1.2
```

```
javax.swing.JColorChooser 1.2
```

```
javax.swing.text.JTextComponent 1.2
```

```
javax.swing.JList 1.2
```

```
javax.swing.JTable 1.2
```

```
javax.swing.JTree 1.2
```

- **void setDragEnabled(boolean b) 1.4**

Включает или отключает перетаскивание данных из соответствующего компонента.

11.14.2. Источники перетаскивания

В предыдущем разделе было показано, как пользоваться элементарными средствами перетаскивания, которые поддерживаются в Swing. А в этом разделе поясняется, как настроить любой компонент, чтобы он выполнял функции источника перетаскивания. Что же касается приемников перетаскивания, то они будут рассмотрены в следующем разделе, где также приводится пример реализации компонента, выполняющего одновременно функции источника и приемника изображений.

Чтобы настроить поведение компонента Swing на перетаскивание, следует создать новый подкласс, производный от класса TransferHandler. В этом подклассе необходимо сначала переопределить метод getSourceActions(), чтобы он указывал, какие именно действия (копирование, перемещение, создание ярлыка) должен поддерживать компонент, а затем — метод getTransferable(), чтобы он выдавал объект типа Transferable таким же образом, как и при копировании данных в буфер обмена.

В рассматриваемом здесь примере программы изображения должны перетаскиваться из компонента JList, заполняемого пиктограммами изображений (рис. 11.45). В приведенном ниже фрагменте кода показано, каким образом метод createTransferable() реализуется в данной программе. При этом выбранное изображение заключается в оболочку объекта типа ImageTransferable.

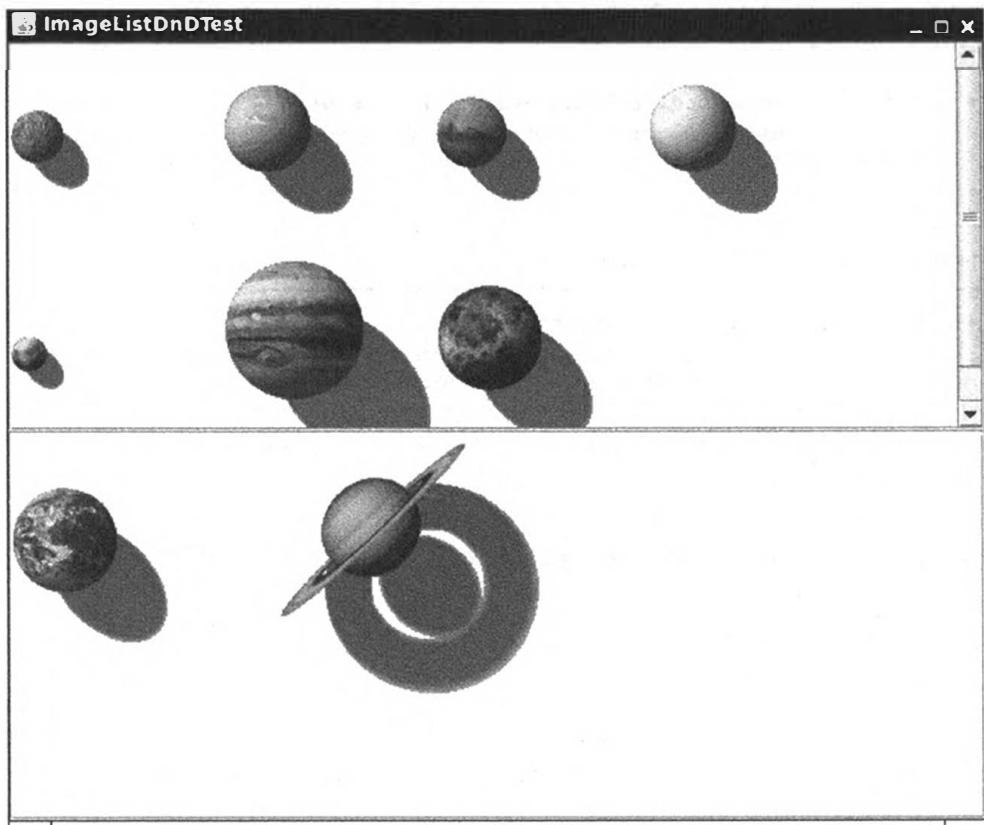


Рис. 11.45. Прикладная программа ImageList, демонстрирующая перетаскивание изображений планет

```
protected Transferable createTransferable(JComponent source)
{
    JList list = (JList) source;
    int index = list.getSelectedIndex();
    if (index < 0) return null;
    ImageIcon icon = (ImageIcon) list.getModel().getElementAt(index);
    return new ImageTransferable(icon.getImage());
}
```

В рассматриваемом здесь примере программы применяется компонент `JList`, который, к счастью, уже предусматривает возможность распознавания жеста перетаскивания. Остается только активизировать соответствующий механизм передачи данных, вызвав метод `setDragEnabled()`. Если же требуется внедрить поддержку перетаскивания в компонент, который не способен распознавать жест перетаскивания, то передачу данных придется инициировать самостоятельно. В качестве примера в приведенном ниже фрагменте кода показано, как инициировать перетаскивание в компоненте `JLabel`.

```
label.addMouseListener(new MouseAdapter()
{
```

```

public void mousePressed(MouseEvent evt)
{
    int mode;
    if ((evt.getModifiers() &
        (InputEvent.CTRL_MASK | InputEvent.SHIFT_MASK)) != 0)
        mode = TransferHandler.COPY;
    else mode = TransferHandler.MOVE;
    JComponent comp = (JComponent) evt.getSource();
    TransferHandler th = comp.getTransferHandler();
    th.exportAsDrag(comp, evt, mode);
}
});

```

В этом фрагменте кода передача данных инициируется, как только пользователь щелкнет на метке. В более сложной реализации перетаскивания можно было бы предусмотреть отслеживание движений мыши при перетаскивании на небольшое расстояние.

По завершении пользователем операции перетаскивания вызывается метод `exportDone()` обработчика передачи данных от источника. Как только пользователь завершит операцию перемещения, в этом методе должно быть произведено удаление переданного объекта. Следовательно, для списка изображений код реализации этого метода должен выглядеть следующим образом:

```

protected void exportDone(JComponent source, Transferable data, int action)
{
    if (action == MOVE)
    {
        JList list = (JList) source;
        int index = list.getSelectedIndex();
        if (index < 0) return;
        DefaultListModel model = (DefaultListModel) list.getModel();
        model.remove(index);
    }
}

```

Таким образом, для превращения компонента в источник перетаскивания достаточно ввести обработчик передачи данных, определяющий следующее:

- какие действия поддерживаются;
- какие данные передаются;
- каким образом исходные данные удаляются после операции перемещения.

Кроме того, если в источник перетаскивания превращается компонент, не входящий в список компонентов, перечисленных в табл. 11.7, придется дополнительно организовать отслеживание движений мыши (жестов) и инициирование передачи данных.

javax.swing.TransferHandler 1.4

- **int getSourceActions(JComponent c)**

Этот метод должен быть переопределен таким образом, чтобы возвращать (в виде поразрядной операции ИЛИ из комбинации констант `COPY`, `MOVE` и `LINK`) сведения о тех действиях, которые источнику разрешено выполнять при перетаскивании из указанного компонента.

javax.swing.TransferHandler 1.4 (окончание)

- **protected Transferable createTransferable(JComponent source)**
Должен быть переопределен таким образом, чтобы создавать объект типа **Transferable** для перетаскиваемых данных.
- **void exportAsDrag(JComponent comp, InputEvent e, int action)**
Начинает операцию перетаскивания из указанного компонента. В качестве параметра **action** может быть указано значение одной из следующих констант: **COPY**, **MOVE** или **LINK**.
- **protected void exportDone(JComponent source, Transferable data, int action)**
Должен быть переопределен таким образом, чтобы настраивать источник перетаскивания соответствующим образом после успешного завершения передачи данных.

11.14.3. Приемники перетаскивания

В этом разделе поясняется, как реализовать приемник перетаскивания. Для примера снова выбран компонент **JList** с пиктограммами изображений. В данном примере требуется предоставить пользователям возможность перетаскивать изображения в список. Чтобы превратить компонент в приемник перетаскивания, следует установить приемник передачи данных типа **TransferHandler** и реализовать методы **canImport()** и **importData()**.



На заметку! Обработчик передачи данных можно ввести и в компонент **JFrame**, который чаще всего применяется в прикладных программах для опускания объектов [например, файлов] после их перетаскивания. К числу допустимых мест опускания перетаскиваемых объектов относятся элементы оформления фрейма и строка меню, но не компоненты внутри фрейма, поскольку у них имеются свои обработчики передачи данных.

Метод **canImport()** вызывается непрерывно, когда пользователь наводит курсор мыши на компонент, выполняющий функции приемника перетаскивания. Если опускание перетаскиваемого объекта разрешается, этот метод возвращает логическое значение **true**. Именно эти сведения и определяют внешний вид пиктограммы курсора, наглядно показывающей, разрешается ли выполнение операции отпускания.

У метода **canImport()** имеется параметр типа **TransferHandler**. **TransferSupport**. Этот параметр позволяет получить сведения о действии опускания перетаскиваемого объекта, выбранным пользователем, о месте опускания и, наконец, о данных, которые требуется передать после опускания. (До версии Java SE 6 существовала другая разновидность метода **canImport()**, позволявшая получать только перечень разновидностей данных.)

В методе **canImport()** имеется также возможность переопределить выбранное пользователем действие опускания перетаскиваемого объекта. Так, если пользователь выбрал в качестве такого действия перемещение, но оно не подходит для удаления оригинала, можно принудить обработчик передачи данных выполнить вместо него копирование.

Ниже приведен типичный пример. Компонент списка изображений готов принимать опускаемые на него списки перетаскиваемых файлов и изображений.

Но если в этот компонент перетаскивается список файлов, то выбранное пользователем действие MOVE (Перемещение) заменяется действием COPY (Копирование), чтобы перетаскиваемые файлы изображений не были удалены.

```
public boolean canImport(TransferSupport support)
{
    if (support.isDataFlavorSupported(DataFlavor.javaFileListFlavor))
    {
        if (support.getUserDropAction() == MOVE)
            support.setDropAction(COPY);
        return true;
    }
    else return support.isDataFlavorSupported(DataFlavor.imageFlavor);
}
```

В более сложной реализации могла бы дополнительно выполняться проверка, действительно ли в файлах содержатся изображения. В таких компонентах Swing, как `JList`, `JTable`, `JTree` и `JTextComponent`, предусмотрено отображение визуальной подсказки для места опускания перетаскиваемого объекта при наведении курсора мыши на поверхность приемника перетаскивания. По умолчанию для обозначения места опускания применяется выделение (если речь идет о компонентах `JList`, `JTable` и `JTree`) либо символ вставки (если речь идет о компоненте `TextComponent`). Но такой способ не совсем удобен для пользователя и применяется по умолчанию только в целях обратной совместимости. Для выбора более подходящей визуальной подсказки следует вызывать метод `setDropMode()`.

Имеется также возможность определить, должны ли опускаемые объекты перезаписывать существующие объекты или быть вставлены между ними. Например, в рассматриваемом здесь примере программы для предоставления пользователю возможности опустить перетаскиваемый объект прямо на существующий элемент (т.е. заменить его) или вставить объект между двумя элементами (рис. 11.46) выполняется приведенный ниже вызов. В табл. 11.8 перечислены все режимы опускания перетаскиваемых объектов, которые поддерживаются компонентами Swing.

```
setDropMode(DropMode.ON_OR_INSERT);
```

Таблица 11.8. Режимы опускания перетаскиваемых объектов

Компонент	Поддерживаемые режимы опускания
<code>JList</code> , <code>JTree</code>	<code>ON</code> , <code>INSERT</code> , <code>ON_OR_INSERT</code> , <code>USE_SELECTION</code>
<code>JTable</code>	<code>ON</code> , <code>INSERT</code> , <code>ON_OR_INSERT</code> , <code>INSERT_ROWS</code> , <code>INSERT_COLS</code> , <code>ON_OR_INSERT_ROWS</code> , <code>ON_OR_INSERT_COLS</code> , <code>USE_SELECTION</code>
<code>JTextComponent</code>	<code>INSERT</code> , <code>USE_SELECTION</code> (фактически перемещает знак вставки, а не выделенный фрагмент текста)

По завершении пользователем операции опускания перетаскиваемого объекта немедленно вызывается метод `importData()`, требующий получения данных из источника перетаскивания. А в результате вызова метода `getTransferable()` для объекта типа `TransferSupport` получается ссылка на объект типа `Transferable`, который предоставляет тот же самый интерфейс, что и для копирования и вставки объектов.

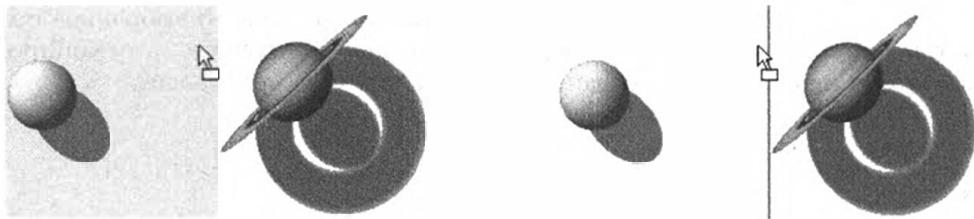


Рис. 11.46. Визуальные подсказки для опускания перетаскиваемого объекта прямо на существующий элемент или вставки объекта между двумя элементами

Для перетаскивания и опускания чаще всего выбирается разновидность данных `DataFlavor.javaFileListFlavor`. Список файлов служит для описания файлов, перетаскиваемых в приемник. Передаваемые данные вводятся в объект типа `List<File>`. Ниже приведен код, требующийся для извлечения файлов из данного списка.

```
 DataFlavor[] flavors = transferable.getTransferDataFlavors();
 if (Arrays.asList(flavors).contains(DataFlavor.javaFileListFlavor))
 {
    List<File> fileList = (List<File>)
        transferable.getTransferData(DataFlavor.javaFileListFlavor);
    for (File f : fileList)
    {
        сделать что-нибудь с файлом f;
    }
 }
```

При опускании перетаскиваемого объекта на какой-нибудь из тех компонентов, которые перечислены в табл. 11.8, требуется точно знать место опускания. Чтобы выяснить это место, можно вызвать метод `getDropLocation()` для объекта типа `TransferSupport`. Этот метод возвращает объект подкласса `TransferHandler.DropLocation`. Классы `JList`, `JTable`, `JTree` и `JTextComponent` определяют подклассы, описывающие местоположение в соответствующей модели данных. Например, в списке местоположение описывается целочисленным индексом, тогда как в дереве — в пути по дереву. Ниже приведен код, требующийся для выяснения места опускания перетаскиваемых объектов в списке изображений.

```
 int index;
 if (support.isDrop())
 {
    JList.DropLocation location =
        (JList.DropLocation) support.getDropLocation();
    index = location.getIndex();
 }
 else index = model.size();
```

В подклассе `JList.DropLocation` имеется метод `getIndex()`, возвращающий индекс места опускания. (А в подклассе `JTree.DropLocation` для выполнения той же самой функции предусмотрен метод `getPath()`.)

Метод `importData()` вызывается и при вставке данных в компонент нажатием комбинации клавиш `<Ctrl+V>`. В таком случае метод `getDropLocation()`

генерирует исключение типа `IllegalStateException`. Но чтобы этого не произошло, вставляемые данные вводятся в конце списка, если метод `isDrop()` возвращает логическое значение `false`.

Кроме того, при вставке данных в список, таблицу или дерево следует проверить, требуется ли вставить эти данные между элементами или заменить ими тот элемент, который находится в месте их опускания. В списке такую проверку можно организовать, вызвав метод `isInsert()` из подкласса `JList.DropLocation`. Что же касается остальных компонентов, то сведения о требующихся для них методах и подклассах можно найти в приведенном далее описании прикладного программного интерфейса API.

Таким образом, чтобы превратить компонент в приемник перетаскивания, достаточно ввести обработчик передачи данных, определяющий следующее:

- когда перетаскиваемый элемент может быть принят;
- каким образом должны импортироваться опускаемые данные.

Если поддержка опускания перетаскиваемых объектов внедряется в компонент `JList`, `JTable`, `JTree` или `JTextComponent`, следует также установить соответствующий режим опускания.

В листинге 11.22 приведен исходный код программы, демонстрирующей перетаскивание и опускание объектов. Обратите внимание на то, что класс `ImageList` одновременно выполняет функции источника и приемника перетаскивания. Попробуйте перетащить изображения из одного списка в другой, а также списки файлов изображений из селектора файлов какой-нибудь другой программы.

Листинг 11.22. Исходный код из файла `dndImage/imageListDnDFrame.java`

```
1 package dndImage;
2
3 import java.awt.*;
4 import java.awt.datatransfer.*;
5 import java.io.*;
6 import java.nio.file.*;
7 import java.util.*;
8 import java.util.List;
9 import javax.imageio.*;
10 import javax.swing.*;
11
12 public class ImageListDnDFrame extends JFrame
13 {
14     private static final int DEFAULT_WIDTH = 600;
15     private static final int DEFAULT_HEIGHT = 500;
16
17     private ImageList list1;
18     private ImageList list2;
19
20     public ImageListDnDFrame()
21     {
22         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
23
24         list1 = new ImageList(
25             Paths.get(getClass().getPackage().getName(),
26             "images1"));
```

```
27     list2 = new ImageList(
28         Paths.get(getClass().getPackage().getName(),
29             "images2"));
30
31     setLayout(new GridLayout(2, 1));
32     add(new JScrollPane(list1));
33     add(new JScrollPane(list2));
34 }
35 }
36
36 class ImageList extends JList<ImageIcon>
37 {
38     public ImageList(Path dir)
39     {
40         DefaultListModel<ImageIcon> model =
41             new DefaultListModel<>();
42         try (DirectoryStream<Path> entries =
43             Files.newDirectoryStream(dir))
44         {
45             for (Path entry : entries)
46                 model.addElement(new ImageIcon(entry.toString()));
47         }
48         catch (IOException ex)
49         {
50             ex.printStackTrace();
51         }
52
53         setModel(model);
54         setVisibleRowCount(0);
55         setLayoutOrientation(JList.HORIZONTAL_WRAP);
56         setDragEnabled(true);
57         setDropMode(DropMode.ON_OR_INSERT);
58         setTransferHandler(new ImageListTransferHandler());
59     }
60 }
61
62 class ImageListTransferHandler extends TransferHandler
63 {
64     // поддержка перетаскивания
65
66     public int getSourceActions(JComponent source)
67     {
68         return COPY_OR_MOVE;
69     }
70
71     protected Transferable createTransferable(JComponent source)
72     {
73         ImageList list = (ImageList) source;
74         int index = list.getSelectedIndex();
75         if (index < 0) return null;
76         ImageIcon icon = list.getModel().getElementAt(index);
77         return new ImageTransferable(icon.getImage());
78     }
79
80     protected void exportDone(JComponent source,
81                             Transferable data, int action)
82     {
83         if (action == MOVE)
84         {
85             ImageList list = (ImageList) source;
```

```
86     int index = list.getSelectedIndex();
87     if (index < 0) return;
88     DefaultListModel<?> model =
89         (DefaultListModel<?>) list.getModel();
90     model.remove(index);
91 }
92 }
93
94 // поддержка опускания
95
96 public boolean canImport(TransferSupport support)
97 {
98     if (support.isDataFlavorSupported(
99             DataFlavor.javaFileListFlavor))
100    {
101        if (support.getUserDropAction() == MOVE)
102            support.setDropAction(COPY);
103        return true;
104    }
105    else return support.isDataFlavorSupported(
106            DataFlavor.imageFlavor);
107 }
108
109 public boolean importData(TransferSupport support)
110 {
111     ImageList list = (ImageList) support.getComponent();
112     DefaultListModel<ImageIcon> model =
113         (DefaultListModel<ImageIcon>) list.getModel();
114
115     Transferable transferable = support.getTransferable();
116     List<DataFlavor> flavors = Arrays.asList(
117             transferable.getTransferDataFlavors());
118
119     List<Image> images = new ArrayList<>();
120
121     try
122     {
123         if (flavors.contains(DataFlavor.javaFileListFlavor))
124         {
125             @SuppressWarnings("unchecked") List<File> fileList =
126                 (List<File>) transferable.getTransferData(
127                     DataFlavor.javaFileListFlavor);
128             for (File f : fileList)
129             {
130                 try
131                 {
132                     images.add(ImageIO.read(f));
133                 }
134                 catch (IOException ex)
135                 {
136                     // не удалось прочитать изображение – пропустить
137                 }
138             }
139         }
140         else if (flavors.contains(DataFlavor.imageFlavor))
141         {
142             images.add((Image) transferable.getTransferData(
143                     DataFlavor.imageFlavor));
144         }
145     }
```

```

146     int index;
147     if (support.isDrop())
148     {
149         JList.DropLocation location = (JList.DropLocation)
150                     support.getDropLocation();
151         index = location.getIndex();
152         if (!location.isInsert())
153             model.remove(index); // сменить место
154     }
155     else index = model.size();
156     for (Image image : images)
157     {
158         model.add(index, new ImageIcon(image));
159         index++;
160     }
161     return true;
162 }
163 catch (IOException | UnsupportedFlavorException ex)
164 {
165     return false;
166 }
167 }
168 }
```

`javax.swing.TransferHandler 1.4`**• `boolean canImport(TransferSupport support) 6`**

Этот метод переопределяется таким образом, чтобы указывать, может ли целевой компонент принимать перетаскиваемый объект, определяемый параметром `support`.

• `boolean importData(TransferSupport support) 6`

Этот метод переопределяется таким образом, чтобы выполнять только операцию опускания или вставки, описываемую параметром `support`, и возвращать логическое значение `true`, если импорт завершен успешно.

`javax.swing.JFrame 1.2`**• `void setTransferHandler(TransferHandler handler) 6`**

Устанавливает обработчик передачи данных, предназначенный для обработки только операций опускания и вставки.

`javax.swing.JList 1.2`**`javax.swing.JTable 1.2`****`javax.swing.JTree 1.2`****`javax.swing.text.JTextComponent 1.2`****• `void setDropMode(DropMode mode) 6`**

Устанавливает для режима опускания данного компонента одно из значений, перечисленных в табл. 11.8.

javax.swing.TransferHandler.TransferSupport 6

- **Component getComponent()**
Получает сведения о целевом компоненте в текущей операции передачи данных.
- **DataFlavor[] getDataFlavors()**
Получает сведения о разновидностях передаваемых данных.
- **boolean isDrop()**
Возвращает логическое значение **true**, если текущая операция передачи данных подразумевает опускание, или логическое значение **false**, если она подразумевает вставку.
- **int getUserDropAction()**
Получает сведения о действии, выбранном пользователем для опускания: **MOVE**, **COPY** или **LINK**.
- **getSourceDropActions()**
Получает сведения о тех действиях опускания, которые разрешаются источником перетаскивания.
- **getDropAction()**
- **setDropAction()**
Получают или устанавливают действие опускания в текущей операции передачи данных. Первоначально это действие выбирается пользователем, но затем оно может переопределиться обработчиком передачи данных.
- **DropLocation getDropLocation()**
Получает сведения о месте опускания или генерирует исключение типа **IllegalStateException**, если текущая операция передачи данных не подразумевает опускание.

javax.swing.TransferHandler.DropLocation 6

- **Point getDropPoint()**
Получает сведения о местоположении курсора мыши при опускании в целевом компоненте.

javax.swing.JList.DropLocation 6

- **boolean isInsert()**
Возвращает логическое значение **true**, если данные должны быть вставлены перед указанным местом, или логическое значение **false**, если они должны заменить существующие данные.
- **int getIndex()**
Получает индекс в модели для вставки или замены.

javax.swing.JTable.DropLocation 6

- **boolean isInsertRow()**
- **boolean isInsertColumn()**
Возвращают логическое значение **true**, если данные должны быть вставлены перед строкой или столбцом.

javax.swing.JTable.DropLocation б(окончание)

- int getRow()
 - int getColumn()

Получают индекс строки или столбца в модели для вставки или замены, а если опускание произошло на пустом участке, то значение **-1**.

javax.swing.JTree.DropLocation 6

- `TreePath getPath()`
 - `int getChildIndex()`

Возвращают путь по дереву и индекс того дочернего элемента, который определяет место опускания наряду с режимом опускания целевого компонента, как описано ниже.

Режим опускания	Операция редактирования в дереве
INSERT	Вставить в качестве дочернего элемента по заданному пути перед дочерним элементом по указанному индексу
ON или USE _SELECTION	Заменить данные в пути (индекс дочернего элемента не используется)
INSERT _OR _ON	Если индекс дочернего элемента равен -1, поступить как в режиме ON , а иначе — как в режиме INSERT

javax.swing.text.JTextComponent.DropLocation 6

- int getIndex()

Возвращает индекс, по которому должны быть вставлены данные.

11.15. Интеграция с платформой

И в завершении этой главы рассмотрим несколько средств для более тесной интеграции прикладных программ на Java с конкретной платформой. Средство создания начальных экранов позволяет отображать начальный экран при запуске виртуальной машины Java. А класс `java.awt.Desktop` обеспечивает возможность запуска платформенно-ориентированных приложений вроде используемого по умолчанию браузера или программы электронной почты в прикладных программах на Java. И, наконец, еще одно средство позволяет получать из прикладных программ на Java доступ к системной области и заполнять ее соответствующими пиктограммами таким же образом, как это делают многие платформенно-ориентированные приложения.

11.15.1. Начальные экраны

К числу наиболее распространенных претензий к прикладным программам на Java относится медленный их запуск на выполнение. Виртуальной машине Java всегда требуется какое-то время на загрузку всех требуемых классов, и особенно это касается Swing-программ, для которых ей приходится загружать немало кода из библиотек Swing и AWT. Пользователям не нравятся прикладные программы, при запуске которых им приходится ждать появления первого экрана, и поэтому они могут пытаться запускать подобные программы несколько раз, если им кажется, что сделать это с первой попытки не удалось. Удобным выходом из столь затруднительного положения является *начальный экран*, т.е. небольшое окно, быстро появляющееся в виде заставки и уведомляющее пользователя, что приложение успешно запущено.

Безусловно, окно с начальным экраном можно отобразить в самом начале выполнения метода `main()`. Но ведь выполнение этого метода начинается только после того, как загрузчик классов загрузит все зависимые классы, на что может уйти время. Вместо этого можно предписать виртуальной машине отображать нужное изображение начального экрана сразу же после запуска. Для указания такого изображения имеются два механизма. С одной стороны, можно указать параметр `-splash` в командной строке следующим образом:

```
java -splash:myimage.png MyApp
```

А с другой стороны, можно указать требующееся изображение в манифесте архивного JAR-файла, как показано ниже.

```
Main-Class: MyApp  
SplashScreen-Image: myimage.gif
```

Это изображение будет воспроизведено сразу при запуске прикладной программы на выполнение и автоматически исчезать при появлении первого окна AWT. В качестве начального экрана разрешается использовать любые изображения формата GIF, JPEG или PNG. Поддерживается также анимация (в формате GIF) и прозрачность (в форматах GIF и PNG).

Если ваша прикладная программа готова к выполнению, как только она достигает метода `main()`, можете пропустить остальную часть этого раздела. Но многие прикладные программы на Java имеют модульную архитектуру, где небольшое ядро загружается при запуске целого ряда подключаемых модулей. Характерными тому примерами служат ИСР Eclipse и NetBeans. В этом случае на начальном экране может отображаться ход загрузки подключаемых модулей.

Добиться такого результата можно двумя способами. Первый способ подразумевает воспроизведение прямо на начальном экране, а второй — замену этого экрана фреймом без обрамления с аналогичным содержимым и воспроизведение уже в этом фрейме. Оба способа демонстрируются в рассматриваемом здесь примере программы.

Для воспроизведения прямо на начальном экране сначала необходимо получить ссылку на этот экран и сведения о его графическом контексте и размерах, как показано ниже.

```
SplashScreen splash = SplashScreen.getSplashScreen();
Graphics2D g2 = splash.createGraphics();
Rectangle bounds = splash.getBounds();
```

Затем воспроизведение выполняется обычным образом. По его завершении вызывается метод `update()`, чтобы гарантировать обновление полученного графического изображения. В рассматриваемом здесь примере программы воспроизводится простой индикатор выполнения, как показано ниже и на рис. 11.47, слева.

```
g.fillRect(x, y, width * percent / 100, height);
splash.update();
```

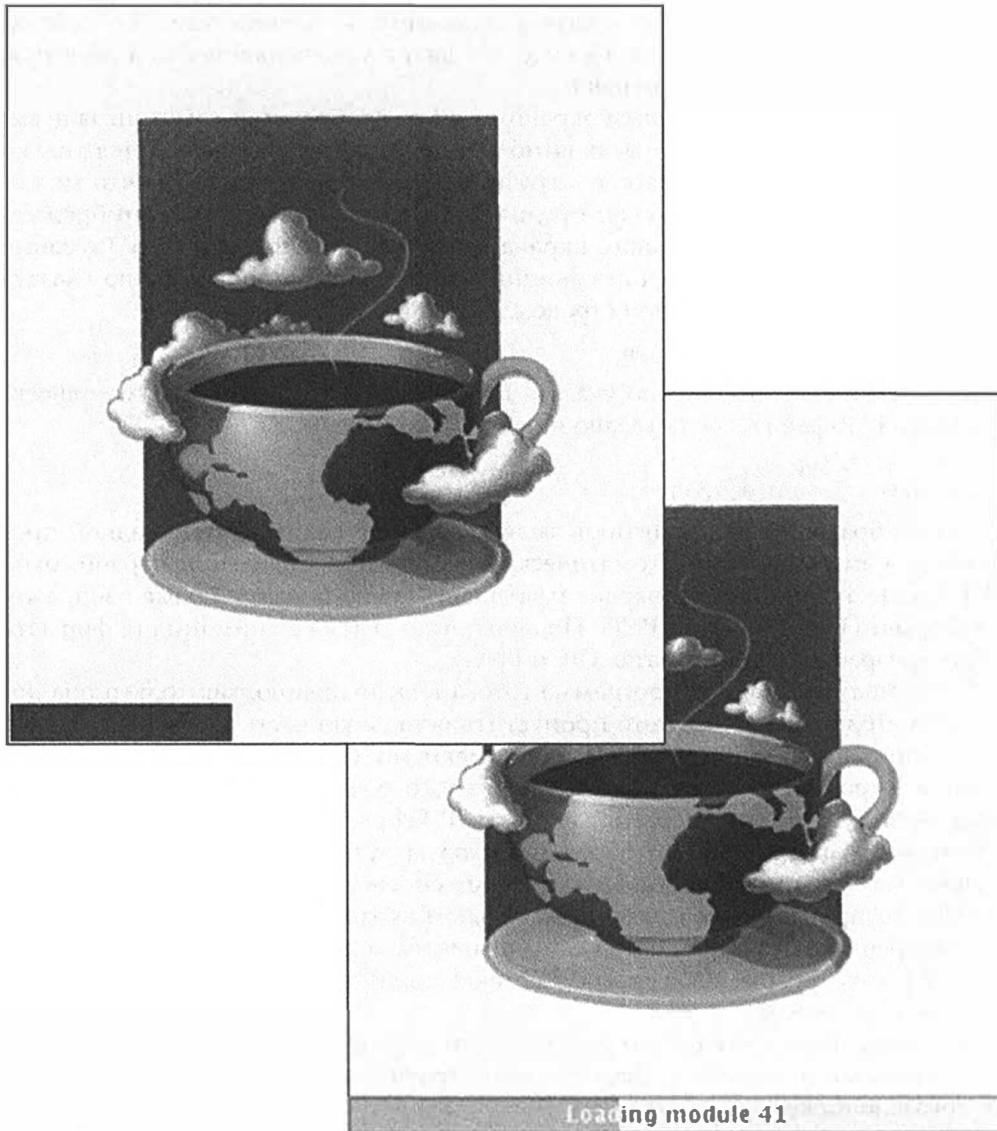


Рис. 11.47. Первоначальный экран и следующее за ним окно без обрамления



На заметку! Начальный экран является одноэлементным объектом (так называемым одиночкой). Создать подобный объект самостоятельно нельзя. Если же начальный экран не указан ни в командной строке, ни в манифесте, метод `getSplashScreen()` возвратит пустое значение `null`.

У воспроизведения прямо на начальном экране имеется следующий существенный недостаток: вычислять позиции всех пикселей утомительно, а воспроизводимый индикатор выполнения обычно не согласуется с собственным индикатором используемой платформы. Поэтому в самом начале выполнения метода `main()` лучше заменить начальный экран последующим окном с теми же разметками и содержимым. Такое окно может содержать любые компоненты Swing.

Именно такой способ демонстрируется в рассматриваемом здесь примере программы из листинга 11.23. На рис. 11.47, справа, показан фрейм без обрамления с панелью, на которой воспроизводится начальный экран и находится компонент `JProgressBar`. Благодаря добавлению такого фрейма открывается полный доступ к прикладному программному интерфейсу API библиотеки Swing, что позволяет без особого труда вводить строки сообщений, не утруждая себя вычислением позиций пикселей. Следует также иметь в виду, что удалять начальный экран не нужно. Он постепенно исчезнет автоматически, как только станет появляться следующее за ним окно.



Внимание! К сожалению, замена начального экрана следующим за ним окном сопровождается заметным мерцанием.

Листинг 11.23. Исходный код из файла `splashScreen/SplashScreenTest.java`

```
1 package splashScreen;
2
3 import java.awt.*;
4 import java.util.List;
5 import javax.swing.*;
6
7 /**
8 * В этой программе демонстрируется прикладной программный
9 * интерфейс API для отображения начального экрана при
10 * запуске прикладной программы на выполнение
11 * @version 1.01 2016-05-10
12 * @author Cay Horstmann
13 */
14 public class SplashScreenTest
15 {
16     private static final int DEFAULT_WIDTH = 300;
17     private static final int DEFAULT_HEIGHT = 300;
18
19     private static SplashScreen splash;
20
21     private static void drawOnSplash(int percent)
22     {
23         Rectangle bounds = splash.getBounds();
24         Graphics2D g = splash.createGraphics();
25         int height = 20;
```

```
26     int x = 2;
27     int y = bounds.height - height - 2;
28     int width = bounds.width - 4;
29     Color brightPurple = new Color(76, 36, 121);
30     g.setColor(brightPurple);
31     g.fillRect(x, y, width * percent / 100, height);
32     splash.update();
33 }
35
36 /**
37 * Этот метод воспроизводит графику на начальном экране
38 */
39 private static void init1()
40 {
41     splash = SplashScreen.getSplashScreen();
42     if (splash == null)
43     {
44         System.err.println("Did you specify a splash image "
45                             + "with -splash or in the manifest?");
46         System.exit(1);
47     }
48
49     try
50     {
51         for (int i = 0; i <= 100; i++)
52         {
53             drawOnSplash(i);
54             Thread.sleep(100); // сымитировать начало работы программы
55         }
56     }
57     catch (InterruptedException e)
58     {
59     }
60 }
61
62 /**
63 * Этот метод отображает фрейм с тем же самым изображением,
64 * что и на начальном экране
65 */
66 private static void init2()
67 {
68     final Image img =
69             new ImageIcon(splash.getImageURL()).getImage();
70
71     final JFrame splashFrame = new JFrame();
72     splashFrame.setUndecorated(true);
73
74     final JPanel splashPanel = new JPanel()
75     {
76         public void paintComponent(Graphics g)
77         {
78             g.drawImage(img, 0, 0, null);
79         }
80     };
81
82     final JProgressBar progressBar = new JProgressBar();
83     progressBar.setStringPainted(true);
```

```
84     splashPanel.setLayout(new BorderLayout());
85     splashPanel.add(progressBar, BorderLayout.SOUTH);
86
87     splashFrame.add(splashPanel);
88     splashFrame.setBounds(splash.getBounds());
89     splashFrame.setVisible(true);
90
91     new SwingWorker<Void, Integer>()
92     {
93         protected Void doInBackground() throws Exception
94         {
95             try
96             {
97                 for (int i = 0; i <= 100; i++)
98                 {
99                     publish(i);
100                     Thread.sleep(100);
101                 }
102             }
103             catch (InterruptedException e)
104             {
105             }
106             return null;
107         }
108
109         protected void process(List<Integer> chunks)
110         {
111             for (Integer chunk : chunks)
112             {
113                 progressBar.setString("Loading module " + chunk);
114                 progressBar.setValue(chunk);
115                 splashPanel.repaint(); // перерисовать изображение,
116                                         // поскольку оно загружается асинхронно
117             }
118         }
119
120         protected void done()
121         {
122             splashFrame.setVisible(false);
123
124             JFrame frame = new JFrame();
125             frame.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
126             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
127             frame.setTitle("SplashScreenTest");
128             frame.setVisible(true);
129         }
130     }.execute();
131 }
132
133 public static void main(String args[])
134 {
135     init1();
136     EventQueue.invokeLater(() -> init2());
137 }
138 }
```

java.awt.SplashScreen 6

- **static SplashScreen getSplashScreen()**

Получает ссылку на начальный экран или пустое значение `null`, если такой экран отсутствует.

- **URL getImageURL()**

- **void setImageURL(URL imageURL)**

Получают или устанавливают URL изображения для начального экрана. Установка изображения приводит к обновлению начального экрана.

- **Rectangle getBounds()**

Получает сведения о границах начального экрана.

- **Graphics2D createGraphics()**

Получает графический контекст для рисования на начальном экране.

- **void update()**

Обновляет отображаемый начальный экран.

- **void close()**

Закрывает начальный экран. Как только появляется первое окно AWT, начальный экран закрывается автоматически.

11.15.2. Запуск настольных приложений

Класс `java.awt.Desktop` позволяет запускать такие используемые по умолчанию настольные приложения, как браузер и программа электронной почты, а также открывать, редактировать и распечатывать файлы с помощью тех приложений, которые зарегистрированы в системе как предназначенные для обработки файлов данного типа. Соответствующий прикладной программный интерфейс API организован очень просто. Сначала вызывается статический метод `isDesktopSupported()`. Если он возвращает логическое значение `true`, это означает, что на данной платформе поддерживается запуск настольных приложений, и тогда вызывается статический метод `getDesktop()` для получения экземпляра класса `Desktop`.

Не все настольные среды поддерживают полностью операции, предусмотренные в рассматриваемом здесь прикладном программном интерфейсе API. Например, в настольной среде Gnome ОС Linux допускается открывать файлы, а распечатывать их нельзя, потому что в ней не поддерживается использование "глаголов" при сопоставлении файлов. Выяснить, какие именно операции поддерживаются на данной платформе, можно, вызвав метод `isSupported()` и передав ему в качестве аргумента одно из значений в перечислении `Desktop.Action`. В рассматриваемом здесь примере программы для этой цели организуются проверки, аналогичные приведенной ниже.

```
if (desktop.isSupported(Desktop.Action.PRINT))
    printButton.setEnabled(true);
```

Для открытия, редактирования или распечатки файла сначала выполняется проверка, поддерживается ли вообще такая операция, и только после этого вызывается метод `open()`, `edit()` или `print()` соответственно. Для запуска браузера передается URI. (Подробнее об URI см. в главе 4.) Можно также вызвать

конструктор URI с символьной строкой, содержащей URL типа http или https. А для запуска используемой по умолчанию программы электронной почты придется составить URI в определенном формате, как показано ниже.

```
mailto:получатель?запрос
```

Здесь получатель обозначает электронный адрес получателя, например president@whitehouse.gov, а запрос — разделенные знаком & пары имя=значение с закодированными знаком % значениями вроде subject=dinner%20RSVP&bcc=putin%40kremvax.ru. (Алгоритм кодирования знаком % действует, по существу, таким же образом, как и описанный в главе 4 алгоритм кодирования URL, только пробел в данном случае кодируется не как +, а как %20.) Данный формат описывается в документе RFC 2368 (<http://www.ietf.org/rfc/rfc2368.txt>). К сожалению, в классе URI не поддерживаются идентификаторы URI в формате mailto, и поэтому составлять и кодировать их приходится вручную.

В рассматриваемом здесь примере программы из листинга 11.24 предоставляется возможность открывать, редактировать или распечатывать любой выби-раемый файл, отображать в окне браузера веб-страницу по любому указанному URL, а также запускать избранную программу электронной почты (рис. 11.48).

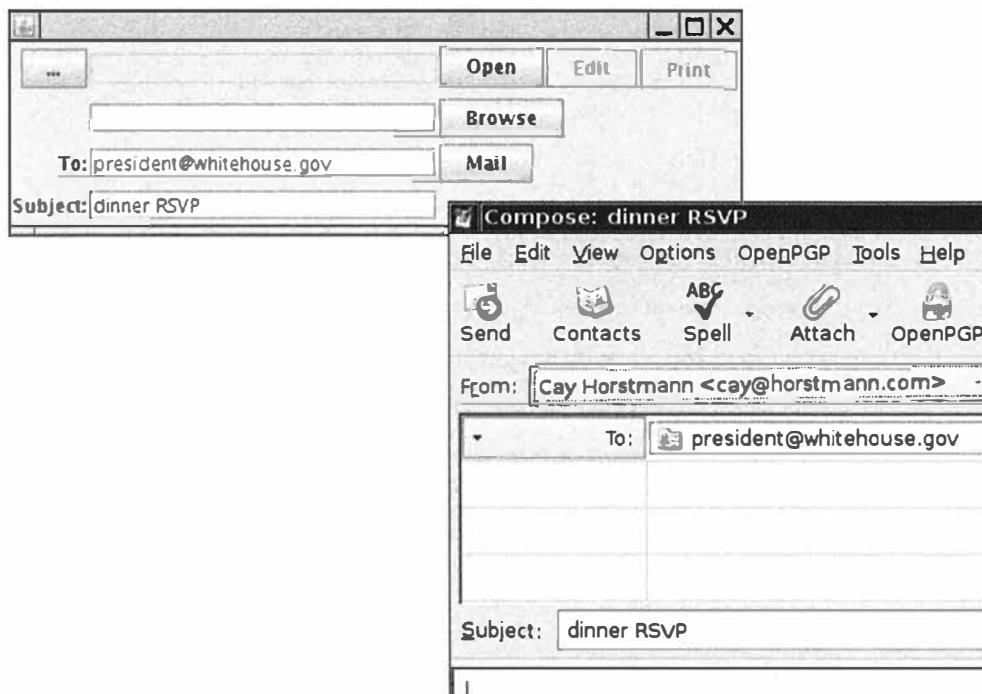


Рис. 11.48. Запуск настольного приложения

Листинг 11.24. Исходный код из файла desktopApp/DesktopAppFrame.java

```
1 package desktopApp;
2
3 import java.awt.*;
4 import java.io.*;
5 import java.net.*;
6
7 import javax.swing.*;
8
9 class DesktopAppFrame extends JFrame
10 {
11     public DesktopAppFrame()
12     {
13         setLayout(new GridBagLayout());
14         final JFileChooser chooser = new JFileChooser();
15         JButton fileChooserButton = new JButton("...");
16         final JTextField fileField = new JTextField(20);
17         fileField.setEditable(false);
18         JButton openButton = new JButton("Open");
19         JButton editButton = new JButton("Edit");
20         JButton printButton = new JButton("Print");
21         final JTextField browseField = new JTextField();
22         JButton browseButton = new JButton("Browse");
23         final JTextField toField = new JTextField();
24         final JTextField subjectField = new JTextField();
25         JButton mailButton = new JButton("Mail");
26
27         openButton.setEnabled(false);
28         editButton.setEnabled(false);
29         printButton.setEnabled(false);
30         browseButton.setEnabled(false);
31         mailButton.setEnabled(false);
32
33         if (Desktop.isDesktopSupported())
34         {
35             Desktop desktop = Desktop.getDesktop();
36             if (desktop.isSupported(Desktop.Action.OPEN))
37                 openButton.setEnabled(true);
38             if (desktop.isSupported(Desktop.Action.EDIT))
39                 editButton.setEnabled(true);
40             if (desktop.isSupported(Desktop.Action.PRINT))
41                 printButton.setEnabled(true);
42             if (desktop.isSupported(Desktop.Action.BROWSE))
43                 browseButton.setEnabled(true);
44             if (desktop.isSupported(Desktop.Action.MAIL))
45                 mailButton.setEnabled(true);
46         }
47
48         fileChooserButton.addActionListener(event ->
49         {
50             if (chooser.showOpenDialog(DesktopAppFrame.this)
51                 == JFileChooser.APPROVE_OPTION)
52                 fileField.setText(chooser.getSelectedFile()
53                                 .getAbsolutePath());
54         });
55
56         openButton.addActionListener(event ->
```

```
57      {
58          try
59          {
60              Desktop.getDesktop().open(chooser.getSelectedFile());
61          }
62          catch (IOException ex)
63          {
64              ex.printStackTrace();
65          }
66      });
67
68     editButton.addActionListener(event ->
69     {
70         try
71         {
72             Desktop.getDesktop().edit(chooser.getSelectedFile());
73         }
74         catch (IOException ex)
75         {
76             ex.printStackTrace();
77         }
78     });
79 });
80
81     printButton.addActionListener(event ->
82     {
83         try
84         {
85             Desktop.getDesktop().print(chooser.getSelectedFile());
86         }
87         catch (IOException ex)
88         {
89             ex.printStackTrace();
90         }
91     });
92
93     browseButton.addActionListener(event ->
94     {
95         try
96         {
97             Desktop.getDesktop().browse(
98                 new URI/browseField.getText());
99         }
100        catch (URISyntaxException | IOException ex)
101        {
102            ex.printStackTrace();
103        }
104    });
105
106    mailButton.addActionListener(event ->
107    {
108        try
109        {
110            String subject = percentEncode(
111                subjectField.getText());
112            URI uri = new URI("mailto:" + toField.getText()
113                + "?subject=" + subject);
114
115            System.out.println(uri);
116            Desktop.getDesktop().mail(uri);
```

```

117      }
118      catch (URISyntaxException | IOException ex)
119      {
120          ex.printStackTrace();
121      }
122  });
123
124 JPanel buttonPanel = new JPanel();
125 ((FlowLayout) buttonPanel.getLayout()).setHgap(2);
126 buttonPanel.add(openButton);
127 buttonPanel.add(editButton);
128 buttonPanel.add(printButton);
129
130 add(fileChooserButton,
131     new GBC(0, 0).setAnchor(GBC.EAST).setInsets(2));
132 add(fileField, new GBC(1, 0).setFill(GBC.HORIZONTAL));
133 add(buttonPanel,
134     new GBC(2, 0).setAnchor(GBC.WEST).setInsets(0));
135 add(browseField, new GBC(1, 1).setFill(GBC.HORIZONTAL));
136 add(browseButton,
137     new GBC(2, 1).setAnchor(GBC.WEST).setInsets(2));
138 add(new JLabel("To:"),
139     new GBC(0, 2).setAnchor(GBC.EAST).setInsets(5, 2, 5, 2));
140 add(toField, new GBC(1, 2).setFill(GBC.HORIZONTAL));
141 add(mailButton,
142     new GBC(2, 2).setAnchor(GBC.WEST).setInsets(2));
143 add(new JLabel("Subject:"),
144     new GBC(0, 3).setAnchor(GBC.EAST).setInsets(5, 2, 5, 2));
145 add(subjectField, new GBC(1, 3).setFill(GBC.HORIZONTAL));
146
147 pack();
148 }
149
150 private static String percentEncode(String s)
151 {
152     try
153     {
154         return URLEncoder.encode(s, "UTF-8")
155             .replaceAll("[+]", "%20");
156     }
157     catch (UnsupportedEncodingException ex)
158     {
159         return null; // кодировка UTF-8 поддерживается всегда
160     }
161 }
162 }

```

java.awt.Desktop 6

- **static boolean isDesktopSupported()**

Возвращает логическое значение **true**, если запуск настольных приложений поддерживается на данной платформе.

- **static Desktop getDesktop()**

Возвращает объект типа **Desktop** для операций запуска настольных приложений. Генерирует исключение типа **UnsupportedOperationException**, если операции запуска настольных приложений не поддерживаются на данной платформе.

java.awt.Desktop 6 (окончание)**• boolean isSupported(Desktop.Action action)**

Возвращает логическое значение `true`, если указанное действие поддерживается. А указать можно одно из следующих действий: `OPEN`, `EDIT`, `PRINT`, `BROWSE` или `MAIL`.

• void open(File file)

Запускает зарегистрированное приложение для открытия указанного файла.

• void edit(File file)

Запускает зарегистрированное приложение для редактирования указанного файла.

• void print(File file)

Распечатывает указанный файл.

• void browse(URI uri)

Запускает используемый по умолчанию браузер с указанным URI.

• void mail()**• void mail(URI uri)**

Запускают используемую по умолчанию программу электронной почты. Во втором варианте допускается заполнение отдельных частей электронного сообщения.

11.15.3. Системная область

Во многих настольных средах имеется область для отображения пиктограмм тех программ, которые выполняются в фоновом режиме и время от времени уведомляют пользователей о тех или иных событиях. В Windows такая область называется *системной*, а отображаемые в ней пиктограммы — *системными*. Такая же терминология принята и в прикладном программном интерфейсе Java API. Типичным примером системной области является монитор, следящий за появлением обновленных версий программного обеспечения. При появлении обновлений монитор может изменить внешний вид отображаемой в системной области пиктограммы или отобразить рядом с ней соответствующее сообщение.

Откровенно говоря, системная область обычно перенасыщена пиктограммами, и поэтому добавление в ней еще одной пиктограммы, как правило, не вызывает у пользователей особого энтузиазма. И рассматриваемая здесь в качестве примера программа, раздающая виртуальные "печенья-гадания", вряд ли станет исключением из этого правила.

Класс `java.awt.SystemTray` является универсальным (независящим от конкретной платформы) проводником в системную область. Как и в рассмотренном выше классе `Desktop`, в нем сначала вызывается статический метод `isSupported()` для проверки, поддерживается ли системная область на локальной платформе Java. Если она поддерживается, то далее вызывается метод `getSystemTray()` для получения одноэлементного объекта типа `SystemTray`.

Самым важным в классе `SystemTray` является метод `add()`, позволяющий добавлять экземпляр класса `TrayIcon`. У любой системной пиктограммы имеются три основных свойства.

- Изображение пиктограммы.
- Всплывающая подсказка, которая появляется при наведении курсора мыши на пиктограмму.
- Всплывающее меню, которое появляется после щелчка правой кнопкой мыши на пиктограмме.

Всплывающее меню вводится в качестве экземпляра класса PopupMenu из библиотеки AWT, представляющего платформенно-ориентированное всплывающее меню, но не меню из библиотеки Swing. А пункты вводятся в меню как экземпляры класса MenuItem из библиотеки AWT. У каждого из пунктов меню, как и у их аналогов из библиотеки Swing, имеется свой приемник действий.

И, наконец, рядом с пиктограммой в системной области можно также выводить различные уведомления для пользователей (рис. 11.49). Для этого достаточно вызвать метод displayMessage() из класса TrayIcon, указав заголовок, текст и тип сообщения:

```
trayIcon.displayMessage("Your Fortune", fortunes.get(index),  
    TrayIcon.MessageType.INFO);
```

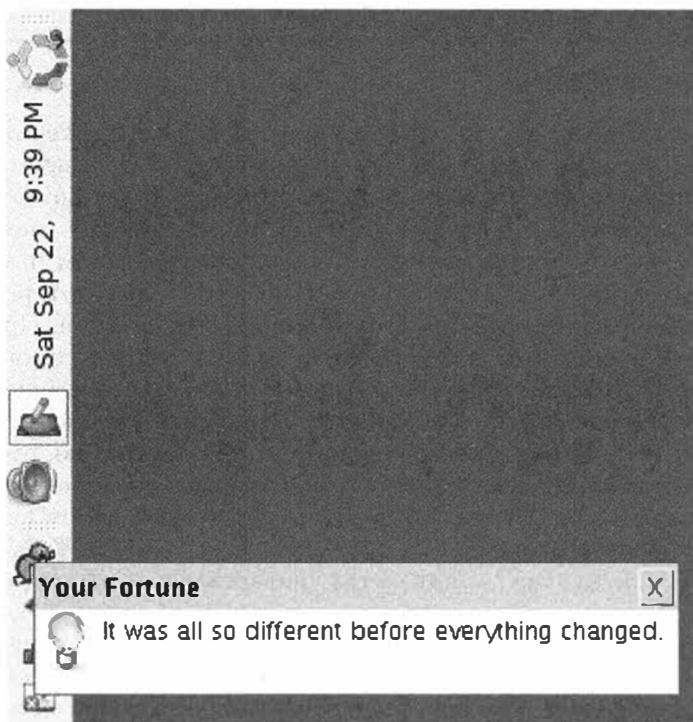


Рис. 11.49. Уведомление от пиктограммы в системной области

В листинге 11.25 приведен исходный код примера прикладной программы, демонстрирующей размещение пиктограммы "печенья-гадания" в системной области. Эта программа считывает файл "печений-гаданий" (из хорошо известной

в UNIX программы fortune), где каждое гадание оканчивается строкой со знаком %, а также отображает новое сообщение каждые десять секунд. Для выхода из данной программы ее пиктограмма в системной области снабжена всплывающим меню с командой завершения работы. Вот если бы все пиктограммы, отображаемые в системной области, были настолько продуманы!

Листинг 11.25. Исходный код из файла systemTray/SystemTrayTest.java

```
1 package systemTray;
2
3 import java.awt.*;
4 import java.io.*;
5 import java.util.*;
6 import java.util.List;
7
8 import javax.swing.*;
9 import javax.swing.Timer;
10
11 /**
12 * В этой программе демонстрируется прикладной интерфейс API
13 * для отображения пиктограмм приложений в системной области
14 * @version 1.02 2016-05-10
15 * @author Cay Horstmann
16 */
17 public class SystemTrayTest
18 {
19     public static void main(String[] args)
20     {
21         SystemTrayApp app = new SystemTrayApp();
22         app.init();
23     }
24 }
25
26 class SystemTrayApp
27 {
28     public void init()
29     {
30         final TrayIcon trayIcon;
31
32         if (!SystemTray.isSupported())
33         {
34             System.err.println("System tray is not supported.");
35             return;
36         }
37
38         SystemTray tray = SystemTray.getSystemTray();
39         Image image = new ImageIcon(getClass()
40             .getResource("cookie.png")).getImage();
41
42         PopupMenu popup = new PopupMenu();
43         MenuItem exitItem = new MenuItem("Exit");
44         exitItem.addActionListener(event -> System.exit(0));
45         popup.add(exitItem);
46
47         trayIcon = new TrayIcon(image, "Your Fortune", popup);
48
49         trayIcon.setImageAutoSize(true);
```

```
50     trayIcon.addActionListener(event ->
51     {
52         trayIcon.displayMessage("How do I turn this off?",
53             "Right-click on the fortune cookie and select Exit.",
54             TrayIcon.MessageType.INFO);
55     });
56
57     try
58     {
59         tray.add(trayIcon);
60     }
61     catch (AWTException e)
62     {
63         System.err.println("TrayIcon could not be added.");
64         return;
65     }
66
67     final List<String> fortunes = readFortunes();
68     Timer timer = new Timer(10000, event ->
69     {
70         int index = (int) (fortunes.size() * Math.random());
71         trayIcon.displayMessage("Your Fortune",
72             fortunes.get(index),
73             TrayIcon.MessageType.INFO);
74     });
75     timer.start();
76 }
77
78 private List<String> readFortunes()
79 {
80     List<String> fortunes = new ArrayList<>();
81     try (InputStream inStream =
82         getClass().getResourceAsStream("fortunes"))
83     {
84         Scanner in = new Scanner(inStream, "UTF-8");
85         StringBuilder fortune = new StringBuilder();
86         while (in.hasNextLine())
87         {
88             String line = in.nextLine();
89             if (line.equals("%"))
90             {
91                 fortunes.add(fortune.toString());
92                 fortune = new StringBuilder();
93             }
94             else
95             {
96                 fortune.append(line);
97                 fortune.append(' ');
98             }
99         }
100    }
101    catch (IOException ex)
102    {
103        ex.printStackTrace();
104    }
105    return fortunes;
106 }
107 }
```

java.awt.SystemTray 6

- **static boolean isSupported()**

Возвращает логическое значение `true`, если на данной платформе поддерживается доступ к системной области.

- **static SystemTray getSystemTray()**

Возвращает объект типа `SystemTray` для получения доступа к системной области. Генерирует исключение типа `UnsupportedOperationException`, если на данной платформе не поддерживается доступ к системной области.

- **Dimension getTrayIconSize()**

Получает размеры пиктограммы в системной области.

- **void add(TrayIcon trayIcon)**

- **void remove(TrayIcon trayIcon)**

Вводит или удаляет пиктограмму из системной области.

java.awt.TrayIcon 6

- **TrayIcon(Image image)**

- **TrayIcon(Image image, String tooltip)**

- **TrayIcon(Image image, String tooltip, PopupMenu popupMenu)**

Создают пиктограмму для отображения в системной области с указанным изображением, всплывающей подсказкой и всплывающим меню.

- **Image getImage()**

- **void setImage(Image image)**

- **String getTooltip()**

- **void setTooltip(String tooltip)**

- **PopupMenu getPopupMenu()**

- **void setPopupMenu(PopupMenu popupMenu)**

Получают или устанавливают изображение, всплывающую подсказку или всплывающее меню для системной пиктограммы.

- **boolean isImageAutoSize()**

- **void setImageAutoSize(boolean autosize)**

Получают или устанавливают свойство `imageAutoSize`. Если это свойство устанавливается, то изображение масштабируется таким образом, чтобы уместиться в области пиктограммы всплывающей подсказки. Если же данное свойство не устанавливается [по умолчанию], то изображение обрезается [если оно слишком велико] или располагается по центру [если оно слишком мало].

- **void displayMessage(String caption, String text, TrayIcon.MessageType messageType)**

Отображает уведомление рядом с системной пиктограммой. Это уведомление может быть одного из следующих видов: `INFO`, `WARNING`, `ERROR` или `NONE`.

java.awt.TrayIcon 6 (окончание)

- **public void addActionListener(ActionListener listener)**
- **public void removeActionListener(ActionListener listener)**

Вводят или удаляют приемник действий, когда вызываемый приемник зависит от конкретной платформы. Обычно вызываются при выполнении одиночного щелчка кнопкой мыши на уведомлении или двойного щелчка на системной пиктограмме.

Вот и подошла к концу эта длинная глава, посвященная расширенным средствам AWT. В завершающей этот том главе будет рассмотрен совсем иной аспект программирования на Java: взаимодействие на одной и той же машине с платформенно-ориентированным кодом, написанным на другом языке программирования.

Платформенно-ориентированные методы

В этой главе...

- ▶ Вызов функции на С из программы на Java
- ▶ Числовые параметры и возвращаемые значения
- ▶ Строковые параметры
- ▶ Доступ к полям
- ▶ Кодирование сигнатур
- ▶ Вызов методов на Java
- ▶ Доступ к элементам массивов
- ▶ Обработка ошибок
- ▶ Применение прикладного программного интерфейса API для вызовов
- ▶ Пример обращения к реестру Windows

Несмотря на все выгоды, которые дает решение применять только Java, возможны ситуации, когда просто необходимо создавать (или использовать) код, написанный на каком-то другом языке. Такой код обычно называется *платформенно-ориентированным*.

В частности, когда язык Java только появился, многие считали, что для ускорения работы критических частей прикладных программ на Java было бы выгоднее использовать язык С или С++. Но на практике такой прием редко оказывался

эффективным. Презентация Java, проведенная на конференции JavaOne в 1996 году, очень ясно это показала. Разработчики криптографической библиотеки из компании Sun Microsystems продемонстрировали, что скорость выполнения криптографических функций, реализованных только в коде Java, оказалась вполне удовлетворительной. Безусловно, она уступала скорости выполнения тех же самых функций, написанных на С, но отличие было не очень значительным. Дело в том, что реализация платформы Java оказалась намного более быстродействующей, чем сетевой ввод-вывод. Именно это и послужило настоящей причиной более низкой производительности.

Разумеется, у перехода на платформенно-ориентированный код имеются свои недостатки. Если часть прикладной программы написана на каком-нибудь другом языке, придется предоставить отдельную специализированную библиотеку для каждой платформы, которую планируется поддерживать. Код, который пишется на С или С++, не предусматривает защиты против некорректного обращения к памяти вследствие неправильно заданного указателя. Это означает, что можно очень легко написать такие платформенно-ориентированные методы, которые будут нарушать работу программы или заражать операционную систему.

Таким образом, платформенно-ориентированный код рекомендуется использовать только в тех случаях, когда это действительно необходимо. В частности, существуют три случая, когда выбор платформенно-ориентированного кода может оказаться правильным.

- Прикладной программе требуется доступ к таким системным функциям или устройствам, которые недоступны на платформе Java.
- Имеется немалое количество проверенного или отлаженного кода на другом языке, а также известны способы его переноса на все требующиеся целевые платформы.
- В результате сопоставительных испытаний обнаружено, что код, написанный на Java, выполняется намного медленнее, чем эквивалентный ему код на другом языке.

В состав платформы Java входит специальный прикладной программный интерфейс API для организации взаимодействия с платформенно-ориентированным кодом на С — JNI (Java Native Interface — платформенно-ориентированный интерфейс Java). И в этой главе обсуждаются вопросы программирования в прикладном интерфейсе JNI.



На заметку C++! Для написания платформенно-ориентированных методов вместо С можно также пользоваться языком С++. Он имеет ряд преимуществ, обеспечивая, например, более строгий контроль соответствия типов и более удобный доступ к функциям JNI. Но в JNI не поддерживается взаимное преобразование классов Java и С++.

12.1 Вызов функций на С из программы на Java

Допустим, имеется некоторая функция, написанная на С, выполняющая что-нибудь полезное, но по той или иной причине ее нежелательно реализовывать

заново на Java. Ради простоты предположим для начала, что это элементарная функция, написанная на C, для вывода приветственного сообщения.

В языке программирования Java для организации вызова платформенно-ориентированного метода служит ключевое слово `native`, а сам метод, очевидно, вводится в класс. В листинге 12.1 показано, как именно это делается. Ключевое слово `native` предупреждает компилятор, что метод будет определяться внешним образом. Разумеется, платформенно-ориентированные методы не должны содержать код, написанный на Java, поэтому объявление метода состоит только из одного заголовка, после которого сразу же следует завершающая точка с запятой. Благодаря этому объявления платформенно-ориентированных методов становятся похожими на объявления абстрактных методов.

Листинг 12.1. Исходный код из файла `helloNative/HelloNative.java`

```
1 /**
2  * @version 1.11 2007-10-26
3  * @author Cay Horstmann
4 */
5 class HelloNative
6 {
7     public static native void greeting();
8 }
```



На заметку! Ради простоты в примерах программ, приведенных в этой главе, пакеты не применяются.

В данном конкретном примере платформенно-ориентированный метод объявляется и как статический (`static`). Платформенно-ориентированные методы могут быть как статическими, так и нестатическими. Их рассмотрение было начато со статического метода, чтобы не усложнять дело передачей параметров.

Класс из приведенного выше примера можно скомпилировать, но если использовать его в программе, то виртуальная машина Java уведомит, что ей неизвестно, как найти функцию `greeting()`, выдав исключение типа `UnsatisfiedLinkError`. Чтобы устранить этот недостаток, следует реализовать платформенно-ориентированный код, т.е. написать соответствующую функцию на C. Имя этой функции должно точно соответствовать ожиданиям виртуальной машины Java. И для этого необходимо соблюдать следующие правила.

1. Использовать полное имя метода из кода Java, т.е. `HelloNative.greeting`. Если класс находится в пакете, добавить перед именем метода имя этого пакета, например `com.horstmann.HelloNative.greeting`.
2. Заменить все точки знаками подчеркивания и присоединить префикс `Java_`, например `Java_HelloNative_greeting` или `Java_com_horstmann_HelloNative_greeting`.
3. Если в имени класса имеются символы, не являющиеся ни буквами, ни цифрами в коде ASCII (например, знак `_` или `$`), или же символы в Юникоде свыше '`\u007F`', заменить их последовательностью символов `_0xxxx`, указав вместо `xxxx` четырехзначное шестнадцатеричное значение заменяемого символа в Юникоде.



На заметку! Если платформенно-ориентированные методы перегружаются, т.е. предоставляется несколько платформенно-ориентированных методов с одинаковым именем, то в конце имени каждого из них следует присоединить два знака подчеркивания и закодированные типы аргументов. (Подробнее о кодировании типов аргументов речь пойдет далее в этой главе.) Так, если предоставляется один платформенно-ориентированный метод `greeting()` и другой платформенно-ориентированный метод `greeting(int repeat)`, первый из них должен именоваться как `Java_HelloNative_greeting_`, а второй — как `Java_HelloNative_greeting_I`.

Разумеется, вручную этого никто не делает, поскольку существует специальная утилита `javah`, способная автоматически формировать правильные имена платформенно-ориентированных функций. Чтобы воспользоваться этой утилитой, необходимо сначала скомпилировать исходный файл. Ниже показано, как это делается на примере исходного файла из листинга 12.1.

```
javac HelloNative.java
```

После этого необходимо вызвать утилиту `javah`, чтобы сформировать с ее помощью заголовочный файл на C из скомпилированного файла класса. Используемый файл утилиты `javah` находится в каталоге `jdk/bin`. Вызывать ее следует вместе с именем класса, т.е. таким же образом, как и компилятор Java. Так, в результате выполнения приведенной ниже команды будет создан заголовочный файл `HelloNative.h`, содержимое которого приведено в листинге 12.2.

```
javah HelloNative
```

Листинг 12.2. Исходный код из файла `helloNative/HelloNative.h`

```

1  /* ЭТОТ ФАЙЛ НЕ РЕДАКТИРУЕТСЯ - он формируется автоматически */
2  #include <jni.h>
3  /* Заголовочный файл для класса HelloNative */
4
5  #ifndef _Included_HelloNative
6  #define _Included_HelloNative
7  #ifdef __cplusplus
8  extern "C" {
9  #endif
10 /*
11  * Класс: HelloNative
12  * Метод: greeting
13  * Сигнатура: ()V
14 */
15 JNIEXPORT void JNICALL Java_HelloNative_greeting
16   (JNIEnv *, jclass);
17
18 #ifdef __cplusplus
19 }
20#endif
21#endif

```

Как следует из листинга 12.2, данный файл содержит объявление функции `Java_HelloNative_greeting`. (Макрокоманды `JNIEXPORT` и `JNICALL` определяются в заголовочном файле `jni.h`. Они обозначают зависящие от компилятора спецификации для экспортации функций из динамически загружаемой библиотеки.)

Теперь остается скопировать прототип функции из заголовочного файла в файл исходного кода и написать для этой функции код реализации, как показано в листинге 12.3. Не обращайте пока что внимания на аргументы `env` и `cl` в этой простой функции. Подробнее об аргументах платформенно-ориентированных функций и методов речь пойдет далее в этой главе.

Листинг 12.3. Исходный код из файла `helloNative/HelloNative.c`

```
1  /*
2   * @version 1.10 1997-07-01
3   * @author Cay Horstmann
4   */
5
6  #include "HelloNative.h"
7  #include <stdio.h>
8
9  JNIEXPORT void JNICALL Java_HelloNative_greeting(
10                      JNIEnv* env, jclass cl)
11 {
12     printf("Hello Native World!\n");
13 }
```



На заметку C++! Для реализации платформенно-ориентированных методов можно использовать и язык C++. Но в этом случае функции, реализующие подобные методы, следует объявлять как `extern "C"`, как выделено полужирным в приведенном ниже примере кода.

```
extern "C"
JNIEXPORT void JNICALL Java_HelloNative_greeting(JNIEnv* env, jclass
cl)
{
    cout << "Hello, Native World!" << endl;
}
```

Это не позволит компилятору C++ "скорректировать" имена методов.

Далее платформенно-ориентированный код на С требуется скомпилировать в динамически загружаемую библиотеку. Этот процесс зависит от используемого компилятора. Например, для вызова компилятора GNU C в ОС Linux соответствующая команда будет выглядеть следующим образом:

```
gcc -fPIC -I jdk/include -I jdk/include/linux -shared
-o libHelloNative.so HelloNative.c
```

Для вызова компилятора С от компании Sun Microsystems в ОС Solaris та же самая команда будет выглядеть так:

```
cc -G -I jdk/include -I jdk/include/solaris -o libHelloNative.so
HelloNative.c
```

А для вызова компилятора С от корпорации Microsoft в ОС Windows эта же команда примет следующий вид:

```
cl -I jdk/include -I jdk/include/win32 -LD HelloNative.c -FeHelloNative.dll
где jdk обозначает каталог, в котором находится комплект JDK.
```



Совет! Если используется компилятор С от корпорации Microsoft, необходимо сначала запустить из командной оболочки командный файл `vcvars32.bat` или `vsvars32.bat`. Этот командный файл автоматически установит путь и переменные окружения, требующиеся для компилятора. Найти этот файл можно в каталоге `c:\Program Files\Microsoft Visual Studio 14.0\Common7\tools` или в каком-нибудь другом каталоге с аналогичным названием. Подробнее об этом см. в документации на применяемую версию Visual Studio.

Кроме того, можно воспользоваться бесплатно распространяемой средой программирования *Cygwin*, доступной для загрузки по адресу <http://www.cygwin.com>. В ее состав входит компилятор GNU C, а также библиотеки для написания кода в стиле Unix под ОС Windows. Если используется среда *Cygwin*, то команда вызова компилятора С будет выглядеть следующим образом (ее нужно ввести неразрывно в одной строке):

```
gcc -mno-cygwin -D __int64="long long" -I jdk/include/
-I jdk/include/win32 -shared -Wl,--add-stdcall-alias
-o HelloNative.dll HelloNative.c
```



На заметку! В версии заголовочного файла `jni_md.h` для Windows содержится следующее объявление типа, характерное для компилятора С от корпорации Microsoft:

```
typedef __int64 jlong;
```

Следовательно, если используется компилятор GNU C, может возникнуть потребность отредактировать данный файл, например, следующим образом:

```
#ifdef __GNUC__
    typedef long long jlong;
#else
    typedef __int64 jlong;
#endif
```

С другой стороны, можно указать параметр командной строки `-D __int64="long long"` при вызове компилятора.

И последнее, что нужно сделать, — ввести в прикладную программу вызов метода `System.loadLibrary()`. Для полной гарантии, что виртуальная машина будет загружать библиотеку перед первым использованием класса, следует организовать статический блок инициализации, как показано в листинге 12.4.

Листинг 12.4. Исходный код из файла `HelloNative/HelloNativeTest.java`

```
1 /**
2  * @version 1.11 2007-10-26
3  * @author Cay Horstmann
4 */
5 class HelloNativeTest
6 {
7     public static void main(String[] args)
8     {
9         HelloNative.greeting();
10    }
11
12    static
13    {
```

```

14     System.loadLibrary("HelloNative");
15 }
16 }
  
```

На рис. 12.1 приведена общая схема платформенно-ориентированного кода. После компиляции и запуска рассматриваемой здесь программы на экране появится сообщение "Hello, Native World!" (Здравствуй, родной мир).

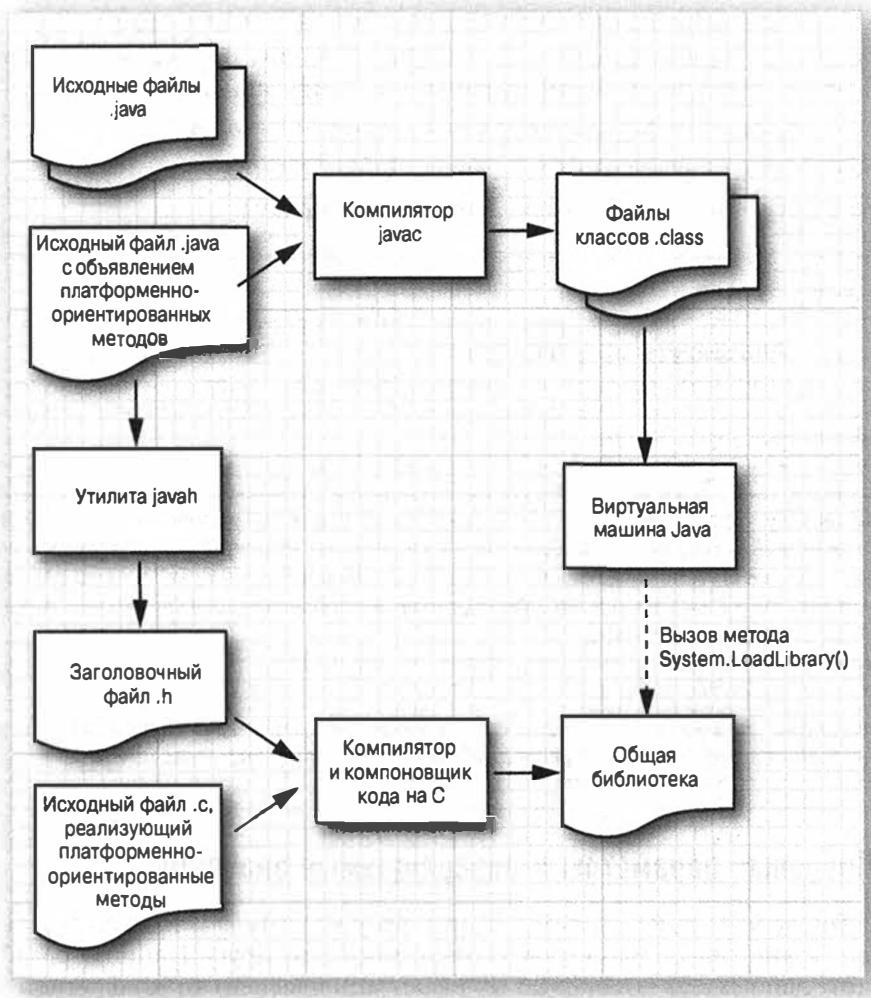


Рис. 12.1. Обработка платформенно-ориентированного кода



На заметку! Пользователям ОС Linux придется включить текущий каталог в путь к библиотеке. Для этого нужно установить переменную окружения **LD_LIBRARY_PATH** следующим образом:
export LD_LIBRARY_PATH=.:\$LD_LIBRARY_PATH
или же установить системное свойство **java.library.path**, как показано ниже.
java -Djava.library.path=. HelloNativeTest

Безусловно, результат не особенно впечатляет. Но если вспомнить, что выводимое сообщение формируется командой, написанной на C, а не на Java, то сразу становится очевидно, что это лишь первые шаги, которые удалось сделать на пути к преодолению пропасти между этими двумя языками программирования!

Таким образом, для связывания платформенно-ориентированного метода с программой на Java необходимо выполнить следующие действия.

1. Объявить платформенно-ориентированный метод в классе Java.
2. Запустить на выполнение утилиту `javah`, чтобы получить заголовочный файл с объявлением платформенно-ориентированного метода на C.
3. Реализовать платформенно-ориентированный метод на C.
4. Разместить полученный код в общей библиотеке.
5. Загрузить общую библиотеку в программу на Java.

`java.lang.System 1.0`

- **`void loadLibrary(String libname)`**

Загружает библиотеку по указанному имени. Библиотека находится в том каталоге, который указан в пути поиска библиотек. Конкретный способ поиска библиотеки зависит от используемой операционной системы.



На заметку! Некоторые общие библиотеки с платформенно-ориентированным кодом требуют выполнения кода инициализации. Любой код инициализации можно разместить в методе `JNI_OnLoad()`. А по завершении своей работы виртуальная машина Java будет аналогичным образом вызывать метод `JNI_Unload()`, если, конечно, предоставить его. Ниже приведены прототипы этих методов.

```
jint JNI_OnLoad(JavaVM* vm, void* reserved);
void JNI_Unload(JavaVM* vm, void* reserved);
```

Метод `JNI_OnLoad()` должен возвратить самую раннюю версию виртуальной машины, которая требуется для нормальной работы библиотеки, например `JNI_VERSION_1_2`.

12.2. Числовые параметры и возвращаемые значения

Для обмена числовыми параметрами между кодами на C и Java следует ясно понимать, какие типы параметров соответствуют друг другу. Например, в C имеются типы данных `int` и `long`, но их реализация зависит от конкретной платформы. На одних платформах тип данных `int` представляет 16-разрядное целочисленное значение, а на других — 32-разрядное. А в Java тип данных `int` всегда представляет только 32-разрядное целочисленное значение. Именно поэтому в JNI и предоставляются такие типы данных, как `jint`, `jlong` и т.д. В табл. 12.1 приведено соответствие типов данных в Java и C.

Таблица 12.1. Типы данных в Java и С

Java	C	Количество байтов
<code>boolean</code>	<code>jboolean</code>	1
<code>byte</code>	<code>jbyte</code>	1
<code>char</code>	<code>jchar</code>	2
<code>short</code>	<code>jshort</code>	2
<code>int</code>	<code>jint</code>	4
<code>long</code>	<code>jlong</code>	8
<code>float</code>	<code>jfloat</code>	4
<code>double</code>	<code>jdouble</code>	8

В заголовочном файле `jni.h` эти типы данных объявляются с помощью операторов `typedef` в качестве эквивалентных типов целевой платформы. В этом же файле определяются также константы `JNI_FALSE = 0` и `JNI_TRUE = 1`.

До появления версии Java SE 5.0 в Java не было непосредственного аналога функции `printf()` на C. В приводимых ниже примерах предполагается, что по той или иной причине используется только старая версия JDK, и поэтому решено реализовать аналогичные функциональные возможности благодаря вызову функции `printf()` на C из платформенно-ориентированного метода. В листинге 12.5 представлен исходный код класса `Printf1`, использующего платформенно-ориентированный метод для вывода числового значения с плавающей точкой, указанной точностью и шириной поля.

Листинг 12.5. Исходный код из файла `printf1/Printf1.java`

```

1  /**
2   * @version 1.10 1997-07-01
3   * @author Cay Horstmann
4  */
5 class Printf1
6 {
7     public static native int print(int width,
8                                     int precision, double x);
9
10    static
11    {
12        System.loadLibrary("Printf1");
13    }
14 }
```

Следует, однако, иметь в виду, что при реализации платформенно-ориентированного метода на C все параметры типа `int` и `double` заменяются параметрами типа `jint` и `jdouble`, как демонстрируется в листинге 12.6.

Листинг 12.6. Исходный код из файла `printf1/Printf1.c`

```

1  /**
2   * @version 1.10 1997-07-01
3   * @author Cay Horstmann
4  */
5 
```

```

6 #include "Printf1.h"
7 #include <stdio.h>
8
9 JNIEXPORT jint JNICALL Java_Printf1_print(JNIEnv* env,
10                               jclass cl, jint width, jint precision, jdouble x)
11 {
12     char fmt[30];
13     jint ret;
14     sprintf(fmt, "%%d.%df", width, precision);
15     ret = printf(fmt, x);
16     fflush(stdout);
17     return ret;
18 }

```

В функции из листинга 12.6 сначала компонуется форматирующая строка "%w.pf" с помощью переменной fmt, а затем вызывается стандартная функция printf(), после чего возвращается количество выведенных символов. В листинге 12.7 приведен исходный код примера программы, где демонстрируется применение класса Printf1.

Листинг 12.7. Исходный код из файла printf1/Printf1Test.java

```

1 /**
2  * @version 1.10 1997-07-01
3  * @author Cay Horstmann
4 */
5 class Printf1Test
6 {
7     public static void main(String[] args)
8     {
9         int count = Printf1.print(8, 4, 3.14);
10        count += Printf1.print(8, 4, count);
11        System.out.println();
12        for (int i = 0; i < count; i++)
13            System.out.print("-");
14        System.out.println();
15    }
16 }

```

12.3. Строковые параметры

А теперь рассмотрим, как обмениваться символьными строками с платформенно-ориентированными методами. Как известно, в Java символьные строки представляют собой последовательности кодовых точек в кодировке UTF-16, а в C — оканчивающиеся нулевым символом последовательности байтов. Иными словами, способы представления символьных строк в этих двух языках программирования заметно отличаются. Поэтому в JNI предоставляются два ряда функций для манипулирования символьными строками: первый из них позволяет преобразовывать символьные строки Java в байтовые последовательности “модифицированного формата UTF-8”, а второй — в массивы символьных значений в кодировке UTF-16, т.е. в массивы типа jchar. (Подробнее о кодировках UTF-8 и UTF-16, а также о “модифицированном формате UTF-8” см. в главе 2.)

Напомним, что в кодировке UTF-8 и “модифицированном формате UTF-8” символы кода ASCII оставляются без изменений, а все остальные символы Юникода кодируются в виде многобайтовых последовательностей.)



На заметку! Стандартный и модифицированный форматы UTF-8 отличаются только дополнительными символами с кодами свыше `0xFFFF`. В стандартной кодировке UTF-8 эти символы кодируются 4-байтовыми последовательностями. А в модифицированном формате символ сначала представляется так называемой “суррогатной парой” в кодировке UTF-16, а затем каждый суррогат кодируется в кодировке UTF-8. В итоге получается последовательность из 6 байт. Такое решение нельзя назвать изящным, но другого выхода нет, ведь спецификация виртуальной машины Java была написана в те времена, когда длина символа в Юникоде ограничивалась 16 битами.

Если в прикладном коде на C уже применяется Юникод, то лучше всего воспользоваться вторым рядом функций преобразования символьных строк. Если же все символьные строки содержат только символы в коде ASCII, то можно воспользоваться функциями преобразования символьных строк в “модифицированный формат UTF-8”.

Платформенно-ориентированный метод со строковым параметром типа `String` фактически получает значение типа `jstring`. А платформенно-ориентированный метод с возвращаемым строковым значением типа `String` должен возвращать значение типа `jstring`. Для чтения и создания объектов типа `jstring` применяются специальные функции JNI. Например, функция `NewStringUTF()` создает новый объект типа `jstring` из массива `char`, содержащего символы в коде ASCII или, что бывает намного чаще, байтовые последовательности, кодированные в “модифицированном формате UTF-8”. Функции JNI вызываются не совсем обычно. Например, вызов функции `NewStringUTF()` выглядит следующим образом:

```
JNIEXPORT jstring JNICALL Java_HelloNative_getGreeting(JNIEnv* env, jclass cl)
{
    jstring jstr;
    char greeting[] = "Hello, Native World\n";
    jstr = (*env)->NewStringUTF(env, greeting);
    return jstr;
}
```



На заметку! Весь код здесь и далее в этой главе приводится на C, если не указано иное.

Во всех вызовах функций JNI используется указатель `env`, который является первым параметром каждого платформенно-ориентированного метода. Он ссылается на таблицу с указателями функций (рис. 12.2). Вследствие этого каждый вызов функции JNI следует предварять префиксом `(*env)->` для разыменования конкретного указателя функции. Более того, указатель `env` является также первым параметром каждой функции JNI.

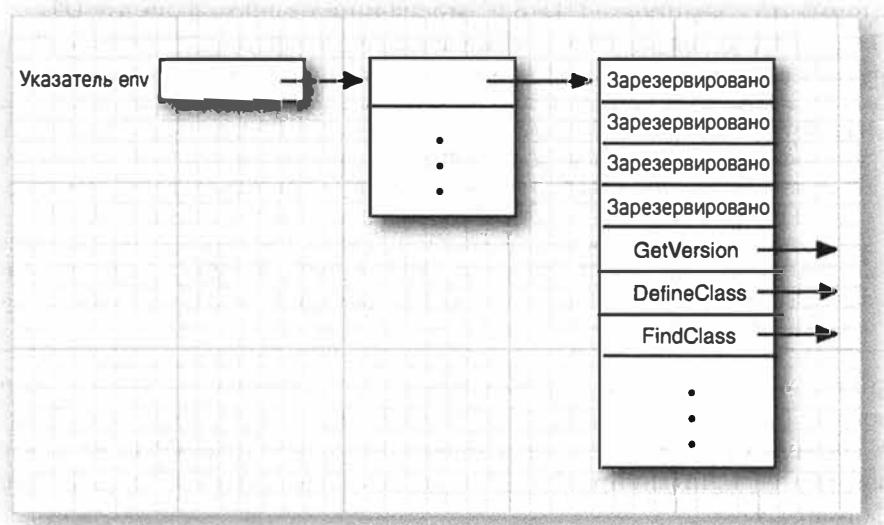


Рис. 12.2. Указатель env



На заметку C++! В языке C++ доступ к функциям JNI организуется проще. В версии класса `JNIEnv` на C++ имеются встраиваемые функции-члены, способные автоматически выполнять поиск указателей функций. Например, вызвать функцию `NewStringUTF()` в C++ можно следующим образом:

```
jstr = env->NewStringUTF(greeting);
```

Обратите внимание на отсутствие указателя `JNIEnv` в списке параметров вызываемого метода.

Функция `NewStringUTF()` позволяет создавать новый объект типа `jstring`. Для считывания содержимого уже существующего объекта типа `jstring` применяется функция `GetStringUTFChars()`, возвращающая указатель `const jbyte*` на описывающие строку символы в модифицированном формате UTF-8. Следует иметь в виду, что некоторые виртуальные машины Java могут использовать данный формат для внутреннего представления символьных строк, из-за чего возможно получение указателя на конкретную символьную строку в Java. А поскольку символьные строки в Java считаются неизменяемыми, то очень важно со всей серьезностью отнестися к обозначению `const` и не пытаться ничего записывать в такой символьный массив. С другой стороны, если виртуальная машина применяет кодировку UTF-16 или UTF-32 для внутреннего представления строк, то вызов функции `GetStringUTFChars()` будет приводить к выделению нового блока памяти и его заполнению эквивалентными символами в модифицированном формате UTF-8.

По завершении манипулирования символьной строкой следует вызвать функцию `ReleaseStringUTFChars()`, чтобы сообщить об этом виртуальной машине, которая должна освободить память, занимаемую данной строкой. Как известно, процесс сборки "мусора" происходит в отдельном потоке исполнения и может прерывать выполнение платформенно-ориентированных методов. Именно поэтому требуется вызывать функцию `ReleaseStringUTFChars()`.

С другой стороны, можно предоставить свой буфер для хранения символов строки, вызвав функции `GetStringRegion()` или `GetStringUTFRegion()`. И наконец, функция `GetStringUTFLength()` возвращает количество символов, необходимых для представления строки в модифицированном формате UTF-8.



На заметку! Описание прикладного программного интерфейса JNI API можно найти по адресу <http://docs.oracle.com/javase/7/docs/technotes/guides/jni/>.

Доступ к символьным строкам в Java из кода на C

- **`jstring NewStringUTF(JNIEnv* env, const char bytes[])`**
Возвращает новый объект символьной строки в Java из оканчивающейся нулевым байтом последовательности байтов в модифицированном формате UTF-8, или значение `NULL`, если сконструировать символьную строку не удается.
- **`jsize GetStringUTFLength(JNIEnv* env, jstring string)`**
Возвращает количество байтов, требующееся для кодирования символьной строки в модифицированном формате UTF-8 (без учета завершающего нулевого байта).
- **`const jbyte* GetStringUTFChars(JNIEnv* env, jstring string, jboolean* isCopy)`**
Возвращает указатель на символьную строку в модифицированном формате UTF-8 или значение `NULL`, если создать символьный массив в таком формате не удается. Этот указатель остается действительным до тех пор, пока не будет вызвана функция `ReleaseStringUTFChars()`. Параметр `isCopy` принимает значение `NULL` или указатель на переменную типа `jboolean`, которая заполняется значением `JNI_TRUE`, если создается копия символьной строки, а иначе — значение `JNI_FALSE`.
- **`void ReleaseStringUTFChars(JNIEnv* env, jstring string, const jbyte bytes[])`**
Извещает виртуальную машину, что из платформенно-ориентированного кода больше не требуется доступ к символьной строке Java через массив `bytes` по указателю, возвращаемому функцией `GetStringUTFChars()`.
- **`void GetStringRegion(JNIEnv* env, jstring string, jsize start, jsize length, jchar* buffer)`**
Копирует последовательность двойных байтов в кодировке UTF-16 из символьной строки в предоставляемый пользователем буфер размером как минимум $2 \times \text{length}$.
- **`void GetStringUTFRegion(JNIEnv* env, jstring string, jsize start, jsize length, jbyte* buffer)`**
Копирует последовательность байтов в модифицированном формате UTF-8 из символьной строки в буфер, предоставленный пользователем. Емкости буфера должно хватить, чтобы вместить все байты. В худшем случае копируется только $3 \times \text{length}$ байтов.
- **`jstring NewString(JNIEnv* env, const jchar chars[], jsize length)`**
Возвращает новый объект символьной строки Java из символьной строки в Юникоде или значение `NULL`, если создать символьную строку Java не удается.

Параметры:

`env`

Указатель на прикладной интерфейс JNI

`chars`

Символьная строка в кодировке

`length`

UTF-16, оканчивающаяся нулевым символом

Количество символов в строке

Доступ к символьным строкам в Java из кода на С (окончание)

- **jsize GetStringLength(JNIEnv* env, jstring string)**
Возвращает количество символов в строке.
- **const jchar* GetStringChars(JNIEnv* env, jstring string, jboolean* isCopy)**
Возвращает указатель на последовательность символов в Юникоде или значение **NULL**, если создать символьный массив в таком формате не удается. Указатель действует до тех пор, пока не будет вызвана функция **ReleaseStringUTFChars()**. Параметр **isCopy** принимает значение **NULL** или указатель на переменную типа **jboolean**, которая заполняется значением **JNI_TRUE**, если создается копия символьной строки, а иначе — значением **JNI_FALSE**.
- **void ReleaseStringChars(JNIEnv* env, jstring string, const jchar chars[])**
Извещает виртуальную машину, что из платформенно-ориентированного кода больше не требуется доступ к символьной строке Java через массив **chars** по указателю, возвращаемому функцией **GetStringChars()**.

А теперь опробуем упомянутые выше функции на практике, разработав класс, способный вызывать функцию **sprintf()** на С. Желательно, чтобы эта функция вызывалась так, как показано в листинге 12.8.

Листинг 12.8. Исходный код из файла printf2/Printf2Test.java

```

1 /**
2  * @version 1.10 1997-07-01
3  * @author Cay Horstmann
4 */
5 class Printf2Test
6 {
7     public static void main(String[] args)
8     {
9         double price = 44.95;
10        double tax = 7.75;
11        double amountDue = price * (1 + tax / 100);
12
13        String s = Printf2.sprintf("Amount due = %8.2f", amountDue);
14        System.out.println(s);
15    }
16 }
```

В листинге 12.9 представлен исходный код класса с платформенно-ориентированным методом **sprint()**.

Листинг 12.9. Исходный код из файла printf2/Printf2.java

```

1 /**
2  * @version 1.10 1997-07-01
3  * @author Cay Horstmann
4 */
5 class Printf2
6 {
```

```
7  public static native String sprint(String format, double x);
8
9  static
10 {
11     System.loadLibrary("Printf2");
12 }
13 }
```

Таким образом, функция на C, с помощью которой форматируется число с плавающей точкой, имеет следующий прототип:

```
JNIEXPORT jstring JNICALL Java_Printf2_sprint(JNIEnv* env, jclass cl,
jstring format, jdouble x)
```

Код, реализующий эту функцию на C, приведен в листинге 12.10. Следует иметь в виду, что для считывания параметра форматирования *format* вызывается функция *GetStringUTFChars()*, для генерирования возвращаемого значения — функция *NewStringUTF()*, а для извещения виртуальной машины, что доступ к символьной строке больше не требуется, — функция *ReleaseStringUTFChars()*.

Листинг 12.10. Исходный код из файла printf2/Printf2.c

```
1  /**
2   * @version 1.10 1997-07-01
3   * @author Cay Horstmann
4  */
5
6 #include "Printf2.h"
7 #include <string.h>
8 #include <stdlib.h>
9 #include <float.h>
10 /**
11  * @param format Символьная строка со спецификатором
12  * формата для функции printf(), например, "%8.2f".
13  * Подстроки "%%" пропускаются
14  * @return Возвращает указатель на спецификатор формата,
15  *         пропуская символ '%', или значение NULL в
16  *         отсутствие однозначного спецификатора формата
17  */
18 char* find_format(const char format[])
19 {
20     char* p;
21     char* q;
22
23     p = strchr(format, '%');
24     while (p != NULL && *(p + 1) == '%')
25         /* пропустить подстроку "%%" */
26         p = strchr(p + 2, '%');
27     if (p == NULL) return NULL;
28     /* проверить единственность символа "%" */
29     p++;
30     q = strchr(p, '%');
31     while (q != NULL && *(q + 1) == '%')
32         /* пропустить подстроку "%%" */
33         q = strchr(q + 2, '%');
```

```

35     if (q != NULL) return NULL; /* символ "%" не единственный */
36     q = p + strspn(p, " -0+#"); /* пропустить признаки */
37     q += strspn(q, "0123456789"); /* пропустить ширину поля */
38     if (*q == '.') { q++; q += strspn(q, "0123456789"); }
39     /* пропустить точность */
40     if (strchr("eEfFgG", *q) == NULL) return NULL;
41     /* это не формат с плавающей точкой */
42     return p;
43 }
44
45 JNIEXPORT jstring JNICALL Java_Printf2_sprint(JNIEnv* env,
46                                     jclass cl, _jstring format, jdouble x)
47 {
48     const char* cformat;
49     char* fmt;
50     jstring ret;
51
52     cformat = (*env)->GetStringUTFChars(env, format, NULL);
53     fmt = find_format(cformat);
54     if (fmt == NULL)
55         ret = format;
56     else
57     {
58         char* cret;
59         int width = atoi(fmt);
60         if (width == 0) width = DBL_DIG + 10;
61         cret = (char*) malloc(strlen(cformat) + width);
62         sprintf(cret, cformat, x);
63         ret = (*env)->NewStringUTF(env, cret);
64         free(cret);
65     }
66     (*env)->ReleaseStringUTFChars(env, format, cformat);
67     return ret;
68 }

```

В данной функции сохранен простой механизм обработки ошибок. Если код форматирования, отвечающий за вывод числового значения с плавающей точкой, не соответствует форме `%w.pc` (где `c` — один из символов `e`, `E`, `f`, `g`, `G`), тогда форматирование числового значения не выполняется. Далее в этой главе будет показано, как организовать в платформенно-ориентированном методе генерирование исключения.

12.4. Доступ к полям

Все рассматривавшиеся до сих пор платформенно-ориентированные методы были статическими с числовыми и строковыми параметрами. А теперь речь пойдет о методах, способных обращаться с объектами произвольного типа. Для примера попробуем реализовать как платформенно-ориентированный метод из класса `Employee`, упоминавшийся в главе 4 первого тома настоящего издания. И хотя это несколько отвлечененный пример, он наглядно показывает, как получается доступ к полям из платформенно-ориентированного метода.

12.4.1. Доступ к полям экземпляра

Чтобы выяснить, как получить доступ к полям экземпляра из платформенно-ориентированного метода, реализуем снова метод `raiseSalary()`. Исходный код этого метода на Java выглядит следующим образом:

```
public void raiseSalary(double byPercent)
{
    salary *= 1 + byPercent / 100;
}
```

Перепишем этот метод как платформенно-ориентированный. В отличие от платформенно-ориентированных методов из предыдущих примеров, данный метод не является статическим. Поэтому в результате выполнения утилиты `javah` получается следующий прототип данного метода:

```
JNIEXPORT void JNICALL Java_Employee_raiseSalary(
    JNIEnv *, jobject, jdouble);
```

Обратите внимание на второй параметр, который теперь имеет тип `jobject`, а не `jclass`. По существу, он является эквивалентом ссылки `this`. Статические методы получают ссылку на класс, а нестатические — на конкретный объект.

Итак, нам требуется доступ к полю `salary` объекта, представленного неявным параметром. В первоначальном варианте связывания кода Java и C в версии Java 1.0 подобная задача решалась очень просто. Программисты могли получить прямой доступ к полям данных объекта. Но для прямого доступа всем виртуальным машинам придется раскрыть внутреннюю структуру данных. Поэтому для получения и установки значений в полях программистам приходится обращаться к специальным функциям JNI.

В рассматриваемом здесь примере поле `salary` относится к типу `double`, поэтому для доступа к нему воспользуемся функциями `GetDoubleField()` и `SetDoubleField()`. Для доступа к полям данных других типов предусмотрены функции `GetIntField()` и `SetIntField()`, `GetObjectField()` и `SetObjectField()` и т.д. При обращении с ними применяется следующий синтаксис:

```
x = (*env)->GetXxxField(env, this_obj, fieldID);
(*env)->SetXxxField(env, this_obj, fieldID, x);
```

где параметр `fieldID` представляет значение специального типа, которое обозначает поле структуры, а суффикс `Xxx` — тип данных Java (например, `Object`, `Boolean`, `Byte` и т.д.). Чтобы получить значение `fieldID`, нужно сначала получить значение, представляющее класс, используя функцию `GetObjectClass()` или `FindClass()`. В частности, функция `GetObjectClass()` возвращает класс произвольного объекта:

```
jclass class_Employee = (*env)->GetObjectClass(env, this_obj);
```

А функция `FindClass()` позволяет указать имя класса в виде символьной строки (вместо точки для разделения пакетов следует использовать знак `/`), как выделено ниже полужирным.

```
jclass class_String = (*env)->FindClass(env, "java/lang/String");
```

Для получения параметра `fieldID` служит функция `GetFieldID()`, при вызове которой указывается имя поля и его *сигнатура*, т.е. кодированное представление его типа. Например, в приведенной ниже строке кода получается идентификатор поля `salary`.

```
jfieldID id_salary = (*env)->GetFieldID(env, class_Employee, "salary", "D");
```

Символьная строка "D" обозначает здесь тип `double`. Более подробно правила кодирования сигнатур полей рассматриваются в следующем разделе.

На первый взгляд такой порядок доступа к полям данных может показаться слишком сложным. Но не следует забывать, что разработчики прикладного программного интерфейса JNI не стремились раскрывать поля данных напрямую, и поэтому им пришлось предоставить функции для установки и получения значений полей. Для сокращения издержек, связанных с применением этих функций, вычисление идентификатора из имени поля выделено в отдельную задачу, поскольку она требует наибольших издержек вычислительных ресурсов. Следовательно, при неоднократном обращении к полю с целью установить или получить его значение идентификатор этого поля вычисляется только один раз.

Итак, принимая во внимание все сказанное выше, реализуем метод `raiseSalary()` как платформенно-ориентированный. Ниже приведен исходный код этого метода, написанный на C.

```
JNIEXPORT void JNICALL Java_Employee_raiseSalary(JNIEnv* env,
                                                 jobject this_obj, jdouble byPercent)
{
    /* получить класс */
    jclass class_Employee = (*env)->GetObjectClass(env, this_obj);
    /* получить идентификатор поля */
    jfieldID id_salary = (*env)->GetFieldID(env, class_Employee,
                                              "salary", "D");
    /* получить значение поля */
    jdouble salary = (*env)->GetDoubleField(env,
                                              this_obj, id_salary);
    salary *= 1 + byPercent / 100;
    /* установить значение поля */
    (*env)->SetDoubleField(env, this_obj, id_salary, salary);
}
```



Внимание! Ссылки на классы действительны только до завершения платформенно-ориентированного метода. Поэтому кешировать значения, возвращаемые функцией `GetObjectClass()`, не удастся. Не пытайтесь сберечь ссылку на класс для последующего использования при вызове платформенно-ориентированного метода. Всякий раз, когда используется платформенно-ориентированный метод, нужно вызывать функцию `GetObjectClass()`. Если же это неприемлемо, ссылку можно зафиксировать с помощью функции `NewGlobalRef()`, например, следующим образом:

```
static jclass class_X = 0;
static jfieldID id_a;

. . .

if (class_X == 0)
{
    jclass cx = (*env)->GetObjectClass(env, obj);
    class_X = (*env)->NewGlobalRef(env, cx);
    id_a = (*env)->GetFieldID(env, cls, "a", ". . .");
}
```

Теперь зафиксированную ссылку на класс и идентификаторы полей можно использовать при последующих вызовах платформенно-ориентированного метода. По завершении всех необходимых операций с классом, доступным по данной ссылке, ее следует удалить с помощью функции `DeleteGlobalRef()`:

```
(*env)->DeleteGlobalRef(env, class_X);
```

В листинге 12.11 приведен исходный код примера программы на Java, а в листинге 12.12 — исходный код класса `Employee`. Что же касается исходного кода платформенно-ориентированного метода `raiseSalary()` на C, то он представлен в листинге 12.13.

Листинг 12.11. Исходный код из файла employee/Employeetest.java

```

1  /**
2   * @version 1.10 1999-11-13
3   * @author Cay Horstmann
4  */
5
6 public class EmployeeTest
7 {
8     public static void main(String[] args)
9     {
10     Employee[] staff = new Employee[3];
11
12     staff[0] = new Employee("Harry Hacker", 35000);
13     staff[1] = new Employee("Carl Cracker", 75000);
14     staff[2] = new Employee("Tony Tester", 38000);
15
16     for (Employee e : staff)
17         e.raiseSalary(5);
18     for (Employee e : staff)
19         e.print();
20 }
21 }
```

Листинг 12.12. Исходный код из файла employee/Employee.java

```

1  /**
2   * @version 1.10 1999-11-13
3   * @author Cay Horstmann
4  */
5
6 public class Employee
7 {
8     private String name;
9     private double salary;
10
11    public native void raiseSalary(double byPercent);
12
13    public Employee(String n, double s)
14    {
15        name = n;
16        salary = s;
17    }
}
```

```

18
19 public void print()
20 {
21     System.out.println(name + " " + salary);
22 }
23
24 static
25 {
26     System.loadLibrary("Employee");
27 }
28
29 }
```

Листинг 12.13. Исходный код из файла employee/Employee.c

```

1 /**
2  * @version 1.10 1999-11-13
3  * @author Cay Horstmann
4 */
5
6 #include "Employee.h"
7
8 #include <stdio.h>
9
10 JNIEXPORT void JNICALL Java_Employee_raiseSalary(JNIEnv* env,
11                                     jobject this_obj, jdouble byPercent)
12 {
13     /* получить класс */
14     jclass class_Employee = (*env)->GetObjectClass(
15         env, this_obj);
16
17     /* получить идентификатор поля */
18     jfieldID id_salary = (*env)->GetFieldID(
19         env, class_Employee, "salary", "D");
20
21     /* получить значение поля */
22     jdouble salary = (*env)->GetDoubleField(env,
23                                                 this_obj, id_salary);
24
25     salary *= 1 + byPercent / 100;
26
27     /* установить значение поля */
28     (*env)->SetDoubleField(env, this_obj, id_salary, salary);
29 }
```

12.4.2. Доступ к статическим полям

Доступ к статическим полям осуществляется аналогично доступу к нестатическим полям. Для этой цели служат функции `GetStaticFieldID()` и `GetStaticXxxField()`/`SetStaticXxxField()`, которые действуют практически так же, как и их нестатические аналоги, но имеют два отличия.

- Вследствие того что объект отсутствует, для получения ссылки на класс следует вызывать функцию `FindClass()` вместо функции `GetObjectClass()`.
- Для доступа к статическому полю следует предоставить класс, а не объект.

В качестве примера ниже показано, как получить ссылку на стандартный поток вывода System.out.

```
/* получить класс */
jclass class_System = (*env)->FindClass(env, "java/lang/System");

/* получить идентификатор поля */
jfieldID id_out = (*env)->GetStaticFieldID(env, class_System, "out",
                                             "Ljava/io/PrintStream;");

/* получить значение поля */
jobject obj_out = (*env)->GetStaticObjectField(env,
                                                 class_System, id_out);
```

Доступ к полям

- **jfieldID GetFieldID(JNIEnv* env, jclass cl, const char name[], const char fieldSignature[])**
Возвращает идентификатор поля в классе.
- **Xxx GetXxxField(JNIEnv* env, jobject obj, jfieldID id)**
Возвращает значение поля. В качестве типа **Xxx** поля может быть указано одно из следующих обозначений: **Object**, **Boolean**, **Byte**, **Char**, **Short**, **Int**, **Long**, **Float** или **Double**.
- **void SetXxxField(JNIEnv* env, jobject obj, jfieldID id, Xxx value)**
Устанавливает новое значение в поле. В качестве типа **Xxx** поля может быть указано одно из следующих обозначений: **Object**, **Boolean**, **Byte**, **Char**, **Short**, **Int**, **Long**, **Float** или **Double**.
- **jfieldID GetStaticFieldID(JNIEnv* env, jclass cl, const char name[], const char fieldSignature[])**
Возвращает идентификатор статического поля в классе.
- **Xxx GetStaticXxxField(JNIEnv* env, jclass cl, jfieldID id)**
Возвращает значение статического поля. В качестве типа **Xxx** поля может быть указано одно из следующих обозначений: **Object**, **Boolean**, **Byte**, **Char**, **Short**, **Int**, **Long**, **Float** или **Double**.
- **void SetStaticXxxField(JNIEnv* env, jclass cl, jfieldID id, Xxx value)**
Устанавливает новое значение в статическом поле. В качестве типа **Xxx** поля может быть указано одно из следующих обозначений: **Object**, **Boolean**, **Byte**, **Char**, **Short**, **Int**, **Long**, **Float** или **Double**.

12.5. Кодирование сигнатур

Для доступа к полям экземпляра и вызова методов, определенных в Java, необходимо знать правила корректирования имен типов данных и сигнатур методов. (Напомним, что сигнтура метода описывает параметры и возвращаемое значение.) Для этой цели используется приведенная ниже схема кодирования.

B	byte
C	char
D	double
F	float
I	int
J	long
Lимя_класса;	тип класса
S	short
V	void
Z	boolean

Для описания типа массива служит знак [. Например, массив символьных строк описывается следующим образом:

[Ljava/lang/String;

А двумерный массив float[][][] кодируется так:

[[[F

Для полного обозначения сигнатуры метода сначала перечисляются все типы параметров в круглых скобках, а затем указывается возвращаемый тип. Например, сигнатура метода, получающего два целочисленных значения и возвращающего одно целочисленное значение, кодируется следующим образом:

(II) I

Метод `sprint()`, упоминавшийся в разделе 12.3, имеет приведенную ниже скорректированную сигнатуру. Она означает, что данный метод получает значение типа `String` и `double` и возвращает значение типа `String`.

(Ljava/lang/String;D)Ljava/lang/String;

Обратите внимание на то, что точка с запятой в конце выражения `Лимя_класса`; служит признаком окончания определяющего тип выражения, а не разделителем параметров. Например, следующий конструктор:

`Employee(java.lang.String, double, java.util.Date)`

имеет такую сигнатуру:

"(Ljava/lang/String;D)Ljava/util/Date;)V"

Обратите внимание на отсутствие разделителя между обозначениями D и `Ljava/util/Date;`, а также на то, что в данной схеме кодирования для разделения имен пакетов и классов требуется указывать знак косой черты (/), а не точки (.). Обозначение V в конце сигнатуры означает, что данный конструктор ничего не возвращает, поскольку возвращаемое значение имеет тип void. И хотя возвращаемый тип для конструкторов в Java не указывается, обозначение V все равно должно добавляться в конце сигнатуры специально для виртуальной машины.



Совет! Для автоматического генерирования сигнатур методов из файлов классов достаточно выполнить команду `javap` с параметром командной строки -s, например, следующим образом:

`javap -s -private Employee`

В итоге на экран будут выведены сигнатуры всех полей и методов указанного класса, как показано ниже.

```
Compiled from "Employee.java"
public class Employee extends java.lang.Object{
private java.lang.String name;
    Signature: Ljava/lang/String;
private double salary;
    Signature: D
public Employee(java.lang.String, double);
    Signature: (Ljava/lang/String;D)V
public native void raiseSalary(double);
    Signature: (D)V
public void print();
    Signature: ()V
static {};
    Signature: ()V
}
```

 **На заметку!** Не существует никакого рационального объяснения тому, что программисты вынуждены применять именно такую скорректированную схему кодирования сигнатур. Разработчики механизма вызова платформенно-ориентированного кода с таким же успехом могли бы написать функцию, позволяющую сначала считывать сигнатуры в стиле языка программирования Java, например `void(int, java.lang.String)`, а затем кодировать их в любое удобное внутреннее представление. Но, с другой стороны, применение скорректированных сигнатур позволяет приобщиться к таинствам программирования на уровне, близком к виртуальной машине.

12.6. Вызов методов на Java

Итак, вы знаете, как вызывать функции, написанные на C, из прикладного кода, написанного на Java. Именно для этого и служат платформенно-ориентированные методы. А можно ли выполнить обратную операцию, вызвав методы на Java из функций на C, и зачем вообще такая операция может потребоваться? Оказывается, что нередко из платформенно-ориентированного метода требуется запрашивать какую-нибудь службу из переданного ему объекта. Поэтому далее будет показано, как добиться этого сначала для методов экземпляра, а затем для статических методов.

12.6.1. Методы экземпляра

В качестве примера вызова метода экземпляра на Java из платформенно-ориентированного кода усовершенствуем класс `Printf`, введя в него метод, действующий как функция `fprintf()` на C. Следовательно, этот метод должен быть в состоянии выводить отформатированную строку в произвольный поток в виде объекта типа `PrintWriter`. Ниже показано, каким образом этот метод определяется на Java.

```
class Printf3
{
    public native static void fprintf(PrintWriter out,
                                      String s, double x);
    ...
}
```

Сначала выводимая символьная строка составляется в объекте `str` типа `String`, как и в реализованном ранее методе `sprint()`. Затем из функции на С вызывается метод `print()` из класса `PrintWriter`. Любой метод на Java можно вызвать из функции на С в следующей форме:

```
(*env)->CallXxxMethod(env, неявный_параметр, идентификатор_метода,
                        явные_параметры)
```

где `Xxx` обозначает возвращаемый тип, например `Void`, `Int`, `Object` и т.д. Если для доступа к полю требуется его идентификатор, то для вызова метода следует указать его идентификатор. Для получения идентификатора вызываемого метода в прикладном программном интерфейсе JNI предусмотрена функция `GetMethodID()` с параметрами, которые задают класс, имя и сигнатуру метода.

В рассматриваемом здесь примере требуется получить идентификатор метода `print()` из класса `PrintWriter`. В классе `PrintWriter` имеются девять разных методов под одинаковым именем `print`. Поэтому для выбора конкретного варианта следует точно указать параметры и возвращаемое значение. Например, для вызова метода `void print(java.lang.String)` следует закодировать его скорректированную сигнатуру в символьной строке `"(Ljava/lang/String;)V"`, как пояснялось в предыдущем разделе.

Ниже приведен весь код, который выполняет следующие действия для вызова этого метода.

6. Получение класса из неявного параметра.
7. Получение идентификатора метода.
8. Вызов метода.

```
/* получить класс */
class_PrintWriter = (*env)->GetObjectClass(env, out);

/* получить идентификатор метода */
id_print = (*env)->GetMethodID(env, class_PrintWriter, "print",
                                 "(Ljava/lang/String;)V");

/* вызвать метод */
(*env)->CallVoidMethod(env, out, id_print, str);
```

В листинге 12.14 приведен исходный код примера программы на Java, а в листинге 12.15 — исходный код класса `Printf3`. Что же касается платформенно-ориентированного метода `fprint()`, то его исходный код на С представлен в листинге 12.16.



На заметку! Числовые идентификаторы методов и полей выполняют те же функции, что и объекты типа `Method` и `Field` из прикладного программного интерфейса API для рефлексии в Java. Для их взаимного преобразования служат следующие функции:

```

jobject ToReflectedMethod(JNIEnv* env, jclass class,
                           jmethodID methodID);
    // возвращает объект типа Method
methodID FromReflectedMethod(JNIEnv* env, jobject method);
jobject ToReflectedField(JNIEnv* env, jclass class,
                           jfieldID fieldID);
    // возвращает объект типа Field
fieldID FromReflectedField(JNIEnv* env, jobject field);

```

12.6.2. Статические методы

Вызов статических методов из платформенно-ориентированного кода выполняется аналогично вызову методов экземпляра, кроме двух отличий.

В качестве примера рассмотрим следующий вызов статического метода `getProperty()` из платформенно-ориентированного метода:

```
System.getProperty("java.class.path")
```

Из этого метода возвращается символьная строка, содержащая текущий путь к классу. Сначала нужно обнаружить используемый класс. В отсутствие объектов класса `System` придется вызвать функцию `FindClass()` вместо функции `GetObjectClass()`:

```
jclass class_System = (*env)->FindClass(env, "java/lang/System");
```

Из этого метода возвращается символьная строка, содержащая текущий путь к классу. Сначала нужно обнаружить используемый класс. В отсутствие объектов класса `System` придется вызвать функцию `FindClass()` вместо функции `GetObjectClass()`, как показано ниже.

```
jclass class_System = (*env)->FindClass(env, "java/lang/System");
```

Затем следует получить идентификатор статического метода `getProperty()`. Параметр и возвращаемое значение этого метода представлены символьной строкой, поэтому его сигнатура кодируется следующим образом:

```
"(Ljava/lang/String;)Ljava/lang/String;"
```

Далее следует получить идентификатор данного метода:

```
jmethodID id_getProperty = (*env)->GetStaticMethodID(env,
                                                       class_System, "getProperty",
                                                       "(Ljava/lang/String;)Ljava/lang/String;");
```

И наконец, остается сделать вызов данного метода, как показано ниже. Следует, однако, иметь в виду, что функции `CallStaticObjectMethod()` передается объект, представляющий класс.

```
jobject obj_ret = (*env)->CallStaticObjectMethod(env,
                                                   class_System, id_getProperty,
                                                   (*env)->NewStringUTF(env, "java.class.path"));
```

Значение, возвращаемое из данного метода, относится к типу `jobject`. Поэтому для выполнения операций над символьными строками его придется привести к типу `jstring` следующим образом:

```
jstring str_ret = (jstring) obj_ret;
```



На заметку C++! В языке С типы `jstring` и `jclass`, а также типы массивов, которые рассматриваются далее, равнозначны типу `jobject`, поэтому выполнять приведение типов в упомянутой выше строке кода на С необязательно. Но в языке C++ эти типы являются указателями на так называемые "фиктивные" классы, имеющие правильную иерархию наследования. Например, присваивание значения типа `jstring` переменной типа `jobject` допускается без приведения типов. Но в то же время приведение типов обязательно для присваивания значения типа `jobject` переменной типа `jstring`.

12.6.3. Конструкторы

В платформенно-ориентированном методе можно создать новый объект Java, вызвав его конструктор с помощью функции `NewObject()` в следующей форме:

```
jobject obj_new = (*env)->NewObject(env, класс,
                                       идентификатор_метода, параметры_конструктора);
```

Чтобы получить идентификатор метода, следует вызвать функцию `GetMethodID()`, указав имя метода в символьной строке "`<init>`" и закодированную сигнатуру конструктора с возвращаемым типом `void`. В качестве примера ниже показано, каким образом в платформенно-ориентированном методе создается объект типа `FileOutputStream` для потока вывода в файл.

```
const char[] fileName = "...";
jstring str_fileName = (*env)->NewStringUTF(env, fileName);
jclass class_FileOutputStream = (*env)->FindClass(env,
                                                       "java/io/FileOutputStream");
jmethodID id_FileOutputStream = (*env)->GetMethodID(env,
                                                       class_FileOutputStream,
                                                       "<init>", "(Ljava/lang/String;)V");
jobject obj_stream = (*env)->NewObject(env,
                                           class_FileOutputStream, id_FileOutputStream,
                                           str_fileName);
```

Обратите внимание на то, что в сигнатуре конструктора описывается параметр типа `java.lang.String` и возвращаемое значение типа `void`.

12.6.4. Альтернативные вызовы методов

В прикладном программном интерфейсе JNI существует несколько вариантов функций, позволяющих вызывать методы на Java из платформенно-ориентированного кода. И хотя эти функции играют не такую важную роль, как рассмотренные ранее функции, иногда они могут принести большую пользу.

В частности, функции `CallNonvirtualXxxMethod()` принимают неявный параметр, идентификатор метода, объект класса (который должен соответствовать суперклассу задаваемого неявным образом аргумента), а также явные параметры. Они вызывают версию метода в указанном классе в обход обычного механизма динамической диспетчеризации. Все подобные функции имеют варианты с суффиксом `A` или `V`. Они получают явно заданные параметры в массиве или структуре типа `va_list`, определенной в заголовочном файле `stdarg.h` на С.

Листинг 12.14. Исходный код из файла printf3/Printf3Test.java

```

1 import java.io.*;
2 /**
3  * @version 1.10 1997-07-01
4  * @author Cay Horstmann
5  */
6 class Printf3Test
7 {
8     public static void main(String[] args)
9     {
10         double price = 44.95;
11         double tax = 7.75;
12         double amountDue = price * (1 + tax / 100);
13         PrintWriter out = new PrintWriter(System.out);
14         Printf3.fprintf(out, "Amount due = %8.2f\n", amountDue);
15         out.flush();
16     }
17 }
```

Листинг 12.15. Исходный код из файла printf3/Printf3.java

```

1 import java.io.*;
2 /**
3  * @version 1.10 1997-07-01
4  * @author Cay Horstmann
5  */
6 class Printf3
7 {
8     public static native void fprintf(PrintWriter out,
9                                         String format, double x);
10
11    static
12    {
13        System.loadLibrary("Printf3");
14    }
15 }
```

Листинг 12.16. Исходный код из файла printf3/Printf3.c

```

1 /**
2  * @version 1.10 1997-07-01
3  * @author Cay Horstmann
4  */
5 #include "Printf3.h"
6 #include <string.h>
7 #include <stdlib.h>
8 #include <float.h>
9 /**
10  * @param format Символьная строка со спецификатором
11  * формата для функции printf(), например "%8.2f".
12  * Подстроки "%%" пропускаются
13  * @return Возвращает указатель на спецификатор формата,
14  *         пропуская символ '%', или значение NULL в
15  *         отсутствие однозначного спецификатора формата
```

```
16  */
17 char* find_format(const char format[])
18 {
19     char* p;
20     char* q;
21
22     p = strchr(format, '%');
23     while (p != NULL && *(p + 1) == '%')
24         /* пропустить подстроку "%%" */
25     p = strchr(p + 2, '%');
26     if (p == NULL) return NULL;
27     /* проверить единственность символа "%" */
28     p++;
29     q = strchr(p, '%');
30     while (q != NULL && *(q + 1) == '%')
31         /* пропустить подстроку "%%" */
32     q = strchr(q + 2, '%');
33     if (q != NULL) return NULL; /* символ "%" не единственный */
34     q = p + strspn(p, "-0+#"); /* пропустить признаки */
35     q += strspn(q, "0123456789"); /* пропустить ширину поля */
36     if (*q == '.') { q++; q += strspn(q, "0123456789"); }
37     /* пропустить точность */
38     if (strchr("eEfFgG", *q) == NULL) return NULL;
39     /* это не формат с плавающей точкой */
40
41     return p;
42 }
43
44 JNIEXPORT void JNICALL Java_Printf3_fprint(JNIEnv* env,
45                                             jclass cl, jobject out, jstring format, jdouble x)
46 {
47     const char* cformat;
48     char* fmt;
49     jstring str;
50     jclass class_PrintWriter;
51     jmethodID id_print;
52     cformat = (*env)->GetStringUTFChars(env, format, NULL);
53     fmt = find_format(cformat);
54     if (fmt == NULL)
55         str = format;
56     else
57     {
58         char* cstr;
59         int width = atoi(fmt);
60         if (width == 0) width = DBL_DIG + 10;
61         cstr = (char*) malloc(strlen(cformat) + width);
62         sprintf(cstr, cformat, x);
63         str = (*env)->NewStringUTF(env, cstr);
64         free(cstr);
65     }
66     (*env)->ReleaseStringUTFChars(env, format, cformat);
67
68     /* вызвать метод ps.print(str) */
69
70     /* получить класс */
71     class_PrintWriter = (*env)->GetObjectClass(env, out);
72
73     /* получить идентификатор метода */
74     id_print = (*env)->GetMethodID(env, class_PrintWriter,
```

```

75         "print", "(Ljava/lang/String;)V");
76
77     /* вызвать метод */
78     (*env)->CallVoidMethod(env, out, id_print, str);
79 }

```

Вызов методов на Java

- `jmethodID GetMethodID(JNIEnv* env, jclass cl, const char name[], const char methodSignature[])`
Возвращает идентификатор метода в классе.
- `Xxx CallXxxMethod(JNIEnv* env, jobject obj, jmethodID id, args)`
- `Xxx CallXxxMethodA(JNIEnv* env, jobject obj, jmethodID id, jvalue args[])`
- `Xxx CallXxxMethodV(JNIEnv* env, jobject obj, jmethodID id, va_list args)`

Вызывают метод. В качестве возвращаемого типа `Xxx` может быть указано одно из следующих обозначений: `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` или `Double`. Первая функция принимает переменное количество аргументов, присоединяемых после идентификатора метода. Вторая функция получает параметры метода в виде массива значений типа `jvalue`, который имеет вид следующего объединения:

```

typedef union jvalue
{
    jboolean z;
    jbyte b;
    jchar c;
    jshort s;
    jint i;
    jlong j;
    jfloat f;
    jdouble d;
    jobject l;
} jvalue;

```

- Третья функция получает параметры метода в виде структуры `va_list`, определяемой в заголовочном файле `stdarg.h` на С.
- `Xxx CallNonvirtualXxxMethod(JNIEnv* env, jobject obj, jclass cl, jmethodID id, args)`
- `Xxx CallNonvirtualXxxMethodA(JNIEnv* env, jobject obj, jclass cl, jmethodID id, jvalue args[])`
- `Xxx CallNonvirtualXxxMethodV(JNIEnv* env, jobject obj, jclass cl, jmethodID id, va_list args)`
- Вызывают метод в обход механизма динамической диспетчеризации. В качестве возвращаемого типа `Xxx` может быть указано одно из следующих обозначений: `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` или `Double`. Первая функция имеет переменное количество аргументов, присоединяемых после идентификатора метода. Параметры метода передаются второй функции в массиве типа `jvalue`. А третья функция получает параметры в виде структуры `va_list`, определяемой в заголовочном файле `stdarg.h` на С.

Вызов методов на Java (окончание)

- `jmethodID GetStaticMethodID(JNIEnv* env, jclass cl, const char name[], const char methodSignature[])`
Возвращает идентификатор статического метода в классе.
- `Xxx CallStaticXxxMethod(JNIEnv* env, jclass cl, jmethodID id, args)`
- `Xxx CallStaticXxxMethodA(JNIEnv* env, jclass cl, jmethodID id, jvalue args[])`
- `Xxx CallStaticXxxMethodV(JNIEnv* env, jclass cl, jmethodID id, va_list args)`

Вызывают статический метод. В качестве возвращаемого типа `Xxx` может быть указано одно из следующих обозначений: `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` или `Double`. Первая функция имеет переменное количество аргументов, присоединяемых после идентификатора метода. Вторая функция получает параметры метода в виде массива типа `jvalue`, а третья — в виде структуры `va_list`, определяемой в заголовочном файле `stdarg.h` на C.

- `jobject NewObject(JNIEnv* env, jclass cl, jmethodID id, args)`
- `jobject NewObjectA(JNIEnv* env, jclass cl, jmethodID id, jvalue args[])`
- `jobject NewObjectV(JNIEnv* env, jclass cl, jmethodID id, va_list args)`

Вызывают конструктор класса. Идентификатор метода получается в результате вызова функции `GetMethodID()` с именем метода в виде строки "`<init>`" и возвращаемым типом `void`. Первая функция имеет переменное количество аргументов, присоединяемых после идентификатора метода. Вторая функция получает параметры метода в виде массива типа `jvalue`, а третья — в виде структуры `va_list`, определяемой в заголовочном файле `stdarg.h` на C.

12.7. Доступ к элементам массивов

Все типы массивов в Java имеют соответствующие им типы массивов в C, перечисленные в табл. 12.2.

Таблица 12.2. Соответствие типов массивов в Java и C

Java	C
<code>boolean[]</code>	<code>jbooleanArray</code>
<code>byte[]</code>	<code>jbyteArray</code>
<code>char[]</code>	<code>jcharArray</code>
<code>int[]</code>	<code>jintArray</code>
<code>short[]</code>	<code>jshortArray</code>
<code>long[]</code>	<code>jlongArray</code>
<code>float[]</code>	<code>jfloatArray</code>
<code>double[]</code>	<code>jdoubleArray</code>
<code>Object[]</code>	<code>jobjectArray</code>



На заметку C++! В языке С все типы массивов, по существу, являются синонимами типа **jobjest**. А в языке C++ они упорядочены в иерархическую структуру, как показано на рис. 12.3. Тип **jarray** обозначает обобщенный массив.

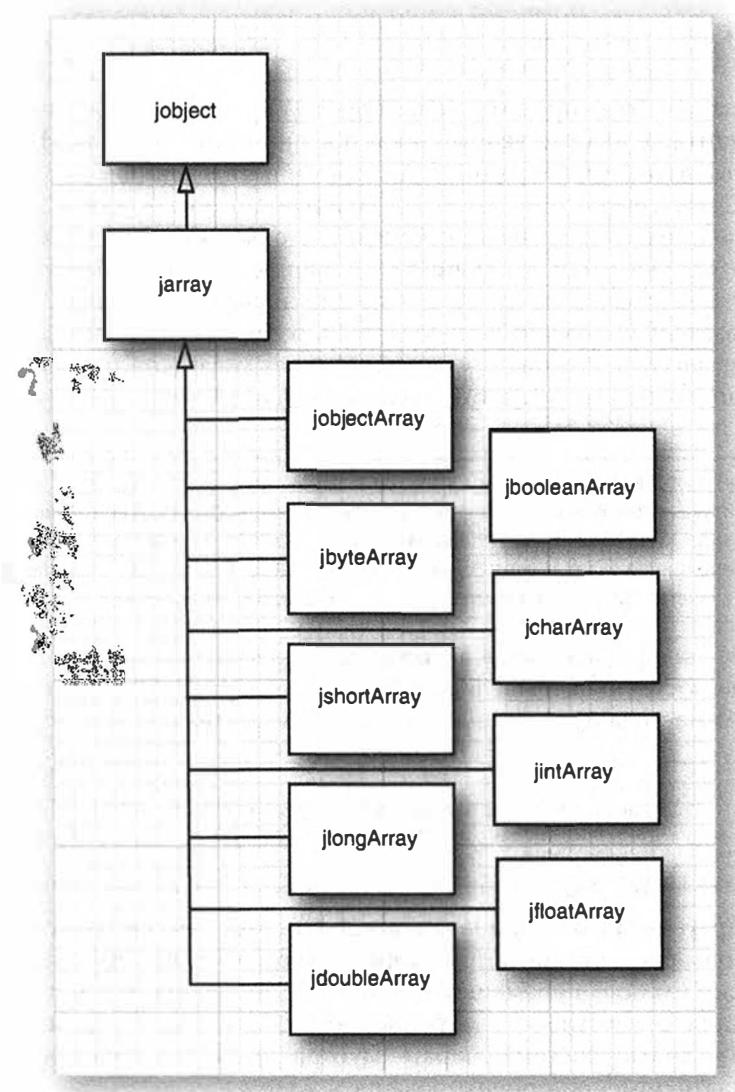


Рис. 12.3. Иерархия наследования типов массивов в C++

Функция `GetArrayLength()` возвращает длину массива, как показано ниже.

```
jarray array = . . .;  
jsize length = (*env)->GetArrayLength(env, array);
```

Порядок доступа к элементам массива зависит от того, хранятся ли в нем объекты или значения примитивных типов (числовые, символьные или логические). Для доступа к элементам массива объектов вызываются функции `GetObjectArrayElement()` и `SetObjectArrayElement()`:

```
 jobjectArray array = . . .;
int i, j;
jobject x = (*env)->GetObjectArrayElement(env, array, i);
(*env)->SetObjectArrayElement(env, array, j, x);
```

Несмотря на всю свою простоту, такой подход крайне неэффективен. Намного лучше получить непосредственный доступ к элементам массива. И это особенно важно для выполнения операций с векторами или матрицами.

Функция `GetXxxArrayElements()` возвращает указатель С на начальный элемент массива. Как и при обращении с обычными символьными строками, если этот указатель больше не нужен, следует вызывать функцию `ReleaseXxxArrayElements()`, где Xxx — примитивный тип данных, а не объект. Используя указатель С, можно непосредственно считывать и записывать значения элементов массива. Но такой указатель может указывать на копию массива, поэтому любые изменения будут отражены в исходном массиве только после вызова функции `ReleaseXxxArrayElements()`.

 **На заметку!** Чтобы выяснить, с каким именно массивом приходится иметь дело: с оригиналом или копией, укажите в качестве третьего параметра функции `GetXxxArrayElements()` переменную типа `jboolean`. Если используется копия массива, то в результате вызова данной функции переменная будет содержать значение `JNI_TRUE`. Если же вас это мало интересует, укажите в качестве третьего параметра функции `GetXxxArrayElements()` значение `NULL`.

Ниже приведен фрагмент кода, в котором все элементы массива типа `double` умножаются на постоянное значение `scaleFactor` типа `double`. Доступ к отдельным элементам массива `a[i]` осуществляется по указателю С.

```
jdoubleArray array_a = . . .;
double scaleFactor = . . .;
double* a = (*env)->GetDoubleArrayElements(env, array_a, NULL);
for (i = 0; i < (*env)->GetArrayLength(env, array_a); i++)
    a[i] = a[i] * scaleFactor;
(*env)->ReleaseDoubleArrayElements(env, array_a, a, 0);
```

Будет ли виртуальная машина Java копировать этот массив, зависит от способа выделения памяти и организации сборки "мусора". Некоторые копирующие системы сборки "мусора" перемещают объекты и обновляют ссылки на объекты. Такая методика не совместима с "закреплением" за массивом определенного места в памяти, поскольку система сборки "мусора" не может обновлять значения указателей в платформенно-ориентированном коде.

 **На заметку!** В виртуальной машине Sun JVM массив значений типа `boolean` представляется в виде массива, содержащего 32-разрядные значения. Функция `GetBooleanArrayElements()` распаковывает его и копирует в массив типа `jboolean`.

Для доступа лишь к нескольким элементам крупного массива служат функции `GetXxxArrayRegion()` и `SetXxxArrayRegion()`. Они копируют из массива Java в массив C и обратно элементы в указанных пределах изменения индексов.

Новые массивы Java в платформенно-ориентированных методах создаются с помощью функции `NewXxxArray()`. Для создания нового массива объектов следует указать его длину, тип элементов массива, а также исходное значение всех элементов (как правило, `NULL`). В приведенном ниже примере кода показано, как это делается.

```
jclass class_Employee = (*env)->FindClass(env, "Employee");
jobjectArray array_e = (*env)->NewObjectArray(env, 100,
                                              class_Employee, NULL);
```

Массивы элементов простых типов устроены проще. Поэтому для них достаточно указать длину массива. Так, в результате выполнения следующей строки кода массив заполняется нулями:

```
jdoubleArray array_d = (*env)->NewDoubleArray(env, 100);
```



На заметку! В версии Java SE 1.4 в прикладном интерфейсе JNI появились следующие три функции:

```
jobject NewDirectByteBuffer(JNIEnv* env, void* address, jlong capacity)
void* GetDirectBufferAddress(JNIEnv* env, jobject buf)
jlong GetDirectBufferCapacity(JNIEnv* env, jobject buf)
```

Непосредственные буферы применяются в пакете `java.nio` для поддержки более эффективных операций ввода-вывода и сведения к минимуму числа операций копирования данных между платформенно-ориентированным кодом и массивами в Java.

Манипулирование массивами Java

- **`jsize GetArrayLength(JNIEnv* env, jarray array)`**
Возвращает количество элементов в массиве.
- **`jobject GetObjectArrayElement(JNIEnv* env, jobjectArray array, jsize index)`**
Возвращает значение указанного элемента массива.
- **`void SetObjectArrayElement(JNIEnv* env, jobjectArray array, jsize index, jobject value)`**
Устанавливает новое значение в элементе массива.
- **`Xxx* GetXxxArrayElements(JNIEnv* env, jarray array, jboolean* isCopy)`**

Выдает указатель C на элементы массива Java. В качестве типа `Xxx` поля может быть указано одно из следующих обозначений: `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` или `Double`. Если указатель больше не требуется, он должен быть передан функции `ReleaseXxxArrayElements()`. Параметр `isCopy` принимает значение `NULL` или указатель на переменную типа `jboolean`, которая заполняется значением `JNI_TRUE`, если создается копия массива, а иначе — значение `JNI_FALSE`.

Манипулирование массивами Java (окончание)

- `void ReleaseXxxArrayElements(JNIEnv* env, jarray array, Xxx elems[], jint mode)`
Уведомляет виртуальную машину Java, что указатель, полученный с помощью функции `GetXxxArrayElements()`, больше не требуется. Параметр `mode` может принимать одно из следующих значений: `0` (освободить буфер `elems` после обновления элементов массива), `JNI_COMMIT` (не освобождать буфер `elems` после обновления элементов массива) или `JNI_ABORT` (освободить буфер `elems` без обновления элементов массива).
- `void GetXxxArrayRegion(JNIEnv* env, jarray array, jint start, jint length, Xxx elems[])`
Копирует элементы из массива Java в массив C. В качестве типа `Xxx` поля может быть указано одно из следующих обозначений: `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` или `Double`.
- `void SetXxxArrayRegion(JNIEnv* env, jarray array, jint start, jint length, Xxx elems[])`
Копирует элементы из массива C в массив Java. В качестве типа `Xxx` поля может быть указано одно из следующих обозначений: `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` или `Double`.

12.8. Обработка ошибок

Применение платформенно-ориентированных методов представляет существенную угрозу нарушения безопасности программ на Java. В языке C не предусмотрена защита от ошибочного указания диапазона массивов, использования некорректных указателей и т.д. Для сохранения целостности программы и виртуальной машины программистам, создающим платформенно-ориентированные методы, следует уделять особое внимание обработке подобных ошибок. Так, если платформенно-ориентированный метод обнаружит какую-нибудь ошибку, которую он не в состоянии обработать самостоятельно, он должен сообщить о ней виртуальной машине Java, которая генерирует соответствующее исключение.

Для создания нового объекта исключения, как правило, вызывается функция `Throw()` или `ThrowNew()`, генерирующая исключение в виртуальной машине Java по завершении платформенно-ориентированного метода. Чтобы воспользоваться функцией `Throw()`, следует вызвать функцию `NewObject()` с целью создать экземпляр класса, производного от класса `Throwable`. Например, в приведенном ниже фрагменте кода создается объект типа `EOFException` и генерируется соответствующее исключение.

```
jclass class_EOFException = (*env)->
    FindClass(env, "java/io/EOFException");
jmethodID id_EOFException = (*env)->
    GetMethodID(env, class_EOFException, "<init>", "()V");
    /* идентификатор конструктора без аргументов */
jthrowable obj_exc = (*env)->
    NewObject(env, class_EOFException, id_EOFException);
(*env)->Throw(env, obj_exc);
```

Но на практике удобнее пользоваться функцией `ThrowNew()`. Она также создает объект исключения, исходя из заданного класса и байтовой последовательности в модифицированном формате UTF-8:

```
(*env)->ThrowNew(env, (*env)->
    FindClass(env, "java/io/EOFException"),
    "Unexpected end of file");
```

Обе функции, `Throw()` и `ThrowNew()`, только передают исключение, но не прерывают поток управления в платформенно-ориентированном методе. Поэтому виртуальная машина Java может генерировать исключение только по завершении платформенно-ориентированного метода. А это означает, что после каждого вызова функции `Throw()` или `ThrowNew()` должен быть указан оператор `return`.



На заметку C++! Если платформенно-ориентированный метод реализуется на C++, в его теле нельзя генерировать объект исключения Java. Теоретически взаимное преобразование исключений в языках C++ и Java возможно, но в настоящее время оно не реализовано. Для генерирования объектов исключений Java в платформенно-ориентированном коде на C++ придется воспользоваться функциями `Throw()` и `ThrowNew()`, а также принять меры, чтобы платформенно-ориентированные методы не генерировали исключения C++.

Как правило, платформенно-ориентированный код не должен заниматься обработкой исключений Java, но подобная ситуация может возникнуть при вызове из платформенно-ориентированной функции метода на Java. Например, функция `SetObjectArrayElement()` генерирует исключение типа `ArrayIndexOutOfBoundsException`, если значение индекса выходит за границы массива, а также исключение типа `ArrayStoreException`, если класс хранимого объекта не является производным от класса элемента массива. В подобных случаях в платформенно-ориентированном методе следует вызвать функцию `ExceptionOccured()`, чтобы определить факт генерирования исключения. Так, в результате приведенного ниже вызова возвращается ссылка на текущий объект исключения. Если же в очереди отсутствуют исключения, ожидающие обработки, то возвращается пустое значение `NULL`.

```
jthrowable obj_exc = (*env)->ExceptionOccurred(env);
```

А если требуется лишь проверить, было ли сгенерировано исключение, не получая ссылку на объект исключения, достаточно сделать следующий вызов:

```
jboolean occurred = (*env)->ExceptionCheck(env);
```

Обычно при генерировании исключения платформенно-ориентированный метод лишь возвращает управление. В итоге виртуальная машина Java может передать исключение соответствующему коду на Java. Но платформенно-ориентированный метод может проанализировать объект исключения и определить, в состоянии ли он обработать его самостоятельно. Если такая обработка возможна, необходимо погасить исключение с помощью следующей функции:

```
(*env)->ExceptionClear(env);
```

В рассматриваемом здесь примере программы реализуется платформенно-ориентированный метод `fprintf()` с некоторыми чрезмерными, на первый взгляд, но весьма полезными на практике мерами для обработки следующих исключений:

- `NullPointerException`, если указатель на форматирующую строку принимает значение `NULL`.
- `IllegalArgumentException`, если форматирующая строка не содержит спецификатор `%`, необходимый для вывода значений типа `double`.
- `OutOfMemoryError`, если произошла ошибка при выполнении функции `malloc()`.

И наконец, чтобы продемонстрировать, каким образом выполняется проверка на наличие исключения при вызове метода на Java из платформенно-ориентированного кода, строка направляется в поток вывода посимвольно, и после вывода каждого символа вызывается функция `ExceptionOccured()`. В листинге 12.17 представлен исходный код платформенно-ориентированного метода, а в листинге 12.18 — исходный код класса, содержащего этот метод. Однако следует иметь в виду, что если при вызове метода `PrintWriter.print()` возникает исключение, то платформенно-ориентированный метод завершает свою работу не сразу — сначала он освобождает буфер `cstr`. Когда же происходит возврат из платформенно-ориентированного метода, виртуальная машина Java снова генерирует исключение. Тестовая программа из листинга 12.19 наглядно демонстрирует, каким образом платформенно-ориентированный метод генерирует исключение, если форматирующая строка оказывается недействительной.

Листинг 12.17. Исходный ход из файла printf4/Printf4.c

```

1  /**
2   * @version 1.10 1997-07-01
3   * @author Cay Horstmann
4   */
5
6  #include "Printf4.h"
7  #include <string.h>
8  #include <stdlib.h>
9  #include <float.h>
10
11 /**
12  * @param format Символьная строка со спецификатором
13  * формата для функции printf(), например, "%8.2f".
14  * Подстроки "%%" пропускаются
15  * @return Возвращает указатель на спецификатор формата,
16  *         пропуская символ '%', или значение NULL в
17  *         отсутствие однозначного спецификатора формата
18 */
19 char* find_format(const char format[])
20 {
21     char* p;
22     char* q;
23
24     p = strchr(format, '%');
25     while (p != NULL && *(p + 1) == '%')
26         /* пропустить подстроку "%%" */
27         p = strchr(p + 2, '%');
28     if (p == NULL) return NULL;
29     /* проверить единственность символа "%" */
30     p++;

```

```
31     q = strchr(p, '%');
32     while (q != NULL && *(q + 1) == '%')
33         /* пропустить подстроку "%%" */
34     q = strchr(q + 2, '%');
35     if (q != NULL) return NULL; /* символ "%" не единственный */
36     q = p + strspn(p, " -0+#"); /* пропустить признаки */
37     q += strspn(q, "0123456789"); /* пропустить ширину поля */
38     if (*q == '.') { q++; q += strspn(q, "0123456789"); }
39     /* пропустить точность */
40     if (strchr("eEfgG", *q) == NULL) return NULL;
41     /* это не формат с плавающей точкой */
42     return p;
43 }
44
45 JNIEXPORT void JNICALL Java_Printf4_fprint(JNIEnv* env,
46                                         jclass cl, jobject out, jstring format, jdouble x)
47 {
48     const char* cformat;
49     char* fmt;
50     jclass class_PrintWriter;
51     jmethodID id_print;
52     char* cstr;
53     int width;
54     int i;
55
56     if (format == NULL)
57     {
58         (*env)->ThrowNew(env, (*env)->FindClass(env,
59                                         "java/lang/NullPointerException"),
60                                         "Printf4.fprint: format is null");
61         return;
62     }
63
64     cformat = (*env)->GetStringUTFChars(env, format, NULL);
65     fmt = find_format(cformat);
66
67     if (fmt == NULL)
68     {
69         (*env)->ThrowNew(env, (*env)->FindClass(env,
70                                         "java/lang/IllegalArgumentException"),
71                                         "Printf4.fprint: format is invalid");
72         return;
73     }
74     width = atoi(fmt);
75     if (width == 0) width = DBL_DIG + 10;
76     cstr = (char*)malloc(strlen(cformat) + width);
77
78     if (cstr == NULL)
79     {
80         (*env)->ThrowNew(env, (*env)->FindClass(env,
81                                         "java/lang/OutOfMemoryError"),
82                                         "Printf4.fprint: malloc failed");
83         return;
84     }
85
86     sprintf(cstr, cformat, x);
87
88     (*env)->ReleaseStringUTFChars(env, format, cformat);
89
90     /* вызвать функцию ps.print(str) */
```

```

91  /* получить класс */
92  class_PrintWriter = (*env)->GetObjectClass(env, out);
93
94  /* получить идентификатор метода */
95  id_print = (*env)->GetMethodID(env, class_PrintWriter,
96                                  "print", "(C)V");
97
98  /* вызвать метод */
99  for (i = 0; cstr[i] != 0 && !(*env)->
101      ExceptionOccurred(env); i++)
102    (*env)->CallVoidMethod(env, out, id_print, cstr[i]);
103
104  free(cstr);
105 }
```

Листинг 12.18. Исходный код из файла printf4/Printf4.java

```

1 import java.io.*;
2
3 /**
4  * @version 1.10 1997-07-01
5  * @author Cay Horstmann
6  */
7 class Printf4
8 {
9     public static native void fprintf(PrintWriter ps,
10                           String format, double x);
11
12     static
13     {
14         System.loadLibrary("Printf4");
15     }
16 }
```

Листинг 12.19. Исходный код из файла printf4/Printf4Test.java

```

1 import java.io.*;
2
3 /**
4  * @version 1.10 1997-07-01
5  * @author Cay Horstmann
6  */
7 class Printf4Test
8 {
9     public static void main(String[] args)
10    {
11        double price = 44.95;
12        double tax = 7.75;
13        double amountDue = price * (1 + tax / 100);
14        PrintWriter out = new PrintWriter(System.out);
15        /* При этом вызове может быть сгенерировано исключение,
16         * если в форматирующй строке отсутствует подстрока "%%" */
17        Printf4.fprintf(out, "Amount due = %%8.2f\n", amountDue);
18        out.flush();
19    }
20 }
```

Обработка исключений в Java

- **jint Throw(JNIEnv* env, jthrowable obj)**
Подготавливает исключение, которое должно генерироваться после выхода из платформенно-ориентированного кода. При удачном исходе возвращает нулевое значение, а иначе — отрицательное.
- **jint ThrowNew(JNIEnv* env, jclass cl, const char msg[])**
Подготавливает исключение типа *cl*, которое должно генерироваться при выходе из платформенно-ориентированного кода. При удачном исходе возвращает нулевое значение, в противном случае — отрицательное. Параметр *msg* служит для создания объекта исключения типа *String* в виде последовательности байтов в "модифицированном формате UTF-8".
- **jthrowable ExceptionOccurred(JNIEnv* env)**
Возвращает объект исключения, если исключение ожидает своей очереди, а иначе — значение *NULL*.
- **jboolean ExceptionCheck(JNIEnv* env)**
Возвращает логическое значение *true*, если исключение все еще ожидает своей очереди.
- **void ExceptionClear(JNIEnv* env)**
Удаляет все исключения, ожидающие своей очереди.

12.9. Применение прикладного программного интерфейса API для вызовов

До сих пор рассматривались только программы на Java, делавшие вызовы на C по двум наиболее вероятным причинам: код на C мог выполняться быстрее или требовался доступ к таким функциональным возможностям, которые были недоступны на платформе Java. А теперь рассмотрим обратную ситуацию, когда имеется программа на C или C++ и требуется организовать из нее несколько вызовов кода на Java. Внедрять виртуальную машину Java в программу на C или C++ позволяет прикладной программный интерфейс API, специально предназначенный для вызовов. Ниже приведен минимальный фрагмент кода, который требуется для инициализации виртуальной машины Java.

```
JavaVMOption options[1];
JavaVMInitArgs vm_args;
JavaVM *jvm;
JNIEnv *env;

options[0].optionString = "-Djava.class.path=.";

memset(&vm_args, 0, sizeof(vm_args));
vm_args.version = JNI_VERSION_1_2;
vm_args.nOptions = 1;
vm_args.options = options;

JNI_CreateJavaVM(&jvm, (void**) &env, &vm_args);
```

Вызов функции *JNI_CreateJavaVM()* приводит к созданию виртуальной машины и заполнению указателя *jvm* на нее, а также указателя *env* на среду

выполнения. Для виртуальной машины Java можно указывать любое количество параметров, увеличивая размер массива `options` и значение в поле `vm_args.nOptions`. Например, выполнение следующей строки кода приведет к отключению динамического компилятора:

```
options[i].optionString = "-Djava.compiler=NONE";
```



Совет! Если возникают неполадки следующего характера: программа завершается аварийно, отказывается инициализировать виртуальную машину Java или не в состоянии загрузить классы, необходимо включить режим отладки. Это делается следующим образом:

```
options[i].optionString = "-verbose:jni";
```

В этом режиме выводятся сообщения с подробными сведениями о ходе инициализации JVM. Если же сообщения о загрузке классов отсутствуют, следует проверить установки как общего пути, так и пути к классам.

После настройки виртуальной машины Java можно приступать к вызову методов на Java способом, описанным в предыдущих разделах, т.е. с помощью узла `env`. А указатель `jvm` следует использовать только в том случае, если требуется вызвать другие функции из прикладного программного интерфейса API. В настоящее время имеется лишь четыре такие функции. Наиболее важной из них является функция `DestroyJavaVM()`, вызываемая для прекращения работы виртуальной машины Java:

```
(*jvm)->DestroyJavaVM(jvm);
```

К сожалению, в Windows стало трудно осуществлять динамическое связывание с функцией `JNI_CreateJavaVM()` из библиотеки `jre/bin/client/jvm.dll` из-за того, что правила связывания в версии Vista изменились, а компания Oracle по-прежнему делает ставку на устаревшую версию библиотеки рабочих программ на C. В рассматриваемом здесь примере программы данное затруднение разрешается путем загрузки этой библиотеки вручную. Тот же самый подход применяется и в утилите `java`. Убедиться в этом можно, проанализировав исходный код из файла `launcher/java_md.c`, находящегося в архивном файле `src.jar`, входящем в состав JDK.

В листинге 12.20 представлен исходный код программы на C, где сначала инициализируется виртуальная машина Java, а затем вызывается метод `main()` из класса `Welcome`, который рассматривался в главе 2 первого тома настоящего издания. (Перед запуском этой программы следует скомпилировать исходный файл `Welcome.java`.)

Листинг 12.20. Исходный код из файла `invocation/InvocationTest.c`

```

1  /**
2   * @version 1.20 2007-10-26
3   * @author Cay Horstmann
4  */
5
6  #include <jni.h>
7  #include <stdlib.h>
8
9  #ifdef _WINDOWS

```

```
10 #include <windows.h>
11 static HINSTANCE loadJVMLibrary(void);
12 typedef jint (JNICALL *CreateJavaVM_t)(JavaVM **, void **,
13                                     JavaVMInitArgs *);
14
15 #endif
16
17
18 int main()
19 {
20     JavaVMOption options[2];
21     JavaVMInitArgs vm_args;
22     JavaVM *jvm;
23     JNIEnv *env;
24     long status;
25
26     jclass class_Welcome;
27     jclass class_String;
28     jobjectArray args;
29     jmethodID id_main;
30
31 #ifdef _WINDOWS
32     HINSTANCE hjvmlib;
33     CreateJavaVM_t createJavaVM;
34 #endif
35
36     options[0].optionString = "-Djava.class.path=.";
37     memset(&vm_args, 0, sizeof(vm_args));
38     vm_args.version = JNI_VERSION_1_2;
39     vm_args.nOptions = 1;
40     vm_args.options = options;
41
42 #ifdef _WINDOWS
43     hjvmlib = loadJVMLibrary();
44     createJavaVM = (CreateJavaVM_t) GetProcAddress(
45                         hjvmlib, "JNI_CreateJavaVM");
46     status = (*createJavaVM)(&jvm, (void **) &env, &vm_args);
47 #else
48     status = JNI_CreateJavaVM(&jvm, (void **) &env, &vm_args);
49 #endif
50
51     if (status == JNI_ERR)
52     {
53         fprintf(stderr, "Error creating VM\n");
54         return 1;
55     }
56
57     class_Welcome = (*env)->FindClass(env, "Welcome");
58     id_main = (*env)->GetStaticMethodID(env, class_Welcome,
59                                         "main", "([Ljava/lang/String;)V");
60
61     class_String = (*env)->FindClass(env, "java/lang/String");
62     args = (*env)->NewObjectArray(env, 0, class_String, NULL);
63     (*env)->CallStaticVoidMethod(env, class_Welcome,
64                                   id_main, args);
65     (*jvm)->DestroyJavaVM(jvm);
66
67     return 0;
```

```
68 }
69
70 #ifdef _WINDOWS
71
71 static int GetStringFromRegistry(HKEY key, const char *name,
72                                 char *buf, jint bufsize)
73 {
74     DWORD type, size;
75
76     return RegQueryValueEx(key, name, 0, &type, 0, &size) == 0
77         && type == REG_SZ
78         && size < (unsigned int) bufsize
79         && RegQueryValueEx(key, name, 0, 0, buf, &size) == 0;
80 }
81
82 static void GetPublicJREHome(char *buf, jint bufsize)
83 {
84     HKEY key, subkey;
85     char version[MAX_PATH];
86     /* найти текущую версию JRE */
87     char *JRE_KEY = "Software\\JavaSoft\\Java Runtime Environment";
88     if (RegOpenKeyEx(HKEY_LOCAL_MACHINE, JRE_KEY, 0,
89                      KEY_READ, &key) != 0)
90     {
91         fprintf(stderr, "Error opening registry key '%s'\n",
92                 JRE_KEY);
93         exit(1);
94     }
95
96     if (!GetStringFromRegistry(key, "CurrentVersion", version,
97                               sizeof(version)))
98     {
99         fprintf(stderr, "Failed reading value of registry key:
100             \n\t%s\\CurrentVersion\n", JRE_KEY);
101         RegCloseKey(key);
102         exit(1);
103     }
104
105     /* Найти каталог, где установлена текущая версия */
106     if (RegOpenKeyEx(key, version, 0, KEY_READ, &subkey) != 0)
107     {
108         fprintf(stderr, "Error opening registry key '%s\\%s'\n",
109                 JRE_KEY, version);
110         RegCloseKey(key);
111         exit(1);
112     }
113
114     if (!GetStringFromRegistry(subkey, "JavaHome", buf, bufsize))
115     {
116         fprintf(stderr, "Failed reading value of registry key:
117             \n\t%s\\%s\\JavaHome\n", 110 JRE_KEY, version);
118         RegCloseKey(key);
119         RegCloseKey(subkey);
120         exit(1);
121     }
122
123     RegCloseKey(key);
124     RegCloseKey(subkey);
```

```

125 }
126
127 static HINSTANCE loadJVMLibrary(void)
128 {
129     HINSTANCE h1, h2;
130     char msวดcdll[MAX_PATH];
131     char javadll[MAX_PATH];
132     GetPublicJREHome(msวดcdll, MAX_PATH);
133     strcpy(javadll, msวดcdll);
134     strncat(mswendll, "\\bin\\msvcr71.dll",
135             MAX_PATH - strlen(mswendll));
136     mswendll[MAX_PATH - 1] = '\0';
137     strncat(javadll, "\\bin\\client\\jvm.dll",
138             MAX_PATH - strlen(javadll));
139     javadll[MAX_PATH - 1] = '\0';
140
141     h1 = LoadLibrary(mswendll);
142     if (h1 == NULL)
143     {
144         fprintf(stderr, "Can't load library msvcr71.dll\n");
145         exit(1);
146     }
147
148     h2 = LoadLibrary(javadll);
149     if (h2 == NULL)
150     {
151         fprintf(stderr, "Can't load library jvm.dll\n");
152         exit(1);
153     }
154     return h2;
155 }
156
157 #endif

```

Чтобы скомпилировать эту программу в ОС Linux, выполните следующую команду:

```
gcc -I jdk/include -I jdk/include/linux -o InvocationTest
    -L jdk/jre/lib/i386/client -ljvm InvocationTest.c
```

А в ОС Solaris выполните для этой же цели такую команду:

```
cc -I jdk/include -I jdk/include/solaris -o InvocationTest
    -L jdk/jre/lib/sparc -ljvm InvocationTest.c
```

Если же требуется скомпилировать данную программу в ОС Windows компилятором от корпорации Microsoft, выполните приведенную ниже команду.

```
cl -D_WINDOWS -I jdk/include -I jdk/include\win32 InvocationTest.c
    jdk\lib\jvm.lib advapi32.lib
```

Убедитесь, что в переменные окружения INCLUDE LIB включен путь к заголовочным файлам и файлам библиотек из прикладного программного интерфейса API для Windows. В среде Cygwin данную программу можно скомпилировать по следующей команде:

```
gcc -D_WINDOWS -mno-cygwin -I jdk/include
    -I jdk/include\win32 -D_int64="long long"
    -I c:\cygwin\usr\include\w32api -o InvocationTest
```

Перед запуском данной программы на выполнение в ОС Linux/UNIX следует удостовериться, что в переменную окружения LD_LIBRARY_PATH включены каталоги для общих библиотек. Так, если в Linux используется командная оболочка bash, то для этой цели следует выполнить такую команду:

```
export LD_LIBRARY_PATH=jdk/jre/lib/i386/client:$LD_LIBRARY_PATH
```

Функции из прикладного программного интерфейса API для вызовов

- **jint JNI_CreateJavaVM(JavaVM** p_jvm, void** p_env, JavaVMInitArgs* vm_args)**

Инициализирует виртуальную машину Java. При удачном завершении возвращает нулевое значение, а иначе — значение **JNI_ERR**.

Параметры:

p_jvm

Указатель на таблицу функций из прикладного программного интерфейса API для вызовов

p_env

Указатель на таблицу функций из прикладного интерфейса JNI

vm_args Аргументы для виртуальной машины

- **jint DestroyJavaVM(JavaVM* jvm)**

Удаляет виртуальную машину Java. При удачном завершении возвращает нулевое значение, в противном случае — отрицательное. Эту функцию следует вызывать по указателю на виртуальную машину, например:

```
(*jvm) ->DestroyJavaVM(jvm);
```

12.10. Пример обращения к реестру Windows

В этом разделе рассматривается пример применения всех перечисленных ранее в этой главе способов обращения с платформенно-ориентированными методами для манипулирования символьными строками, массивами и объектами, вызова конструкторов и обработки исключений. Основная цель данного примера — продемонстрировать создание на платформе Java оболочки для подмножества функций, доступных в прикладном программном интерфейсе API, реализованных на С и предназначенных для работы с реестром ОС Windows. В связи с тем что в рассматриваемом здесь примере программы применяются функции для манипулирования таким специфическим объектом, как реестр Windows, она, разумеется, не может быть перенесена на другие платформы. Именно поэтому в Java не предусмотрено никаких инструментальных средств для работы с реестром, а предполагается, что для этой цели будет использован платформенно-ориентированный код.

12.10.1. Общее представление о реестре Windows

Как известно, реестр представляет собой хранилище конфигурационных данных для операционной системы Windows и прикладных программ. Наличие такого хранилища очень удобно для администрирования, создания резервных

копий и настройки прикладных программ. Однако недостаток реестра заключается в том, что он является также единой точкой отказа: при наличии ошибок в реестре вся операционная система будет работать со сбоями, а возможно, и вообще не загрузится!

В связи с этим пользоваться реестром для хранения конфигурационных параметров прикладных программ на Java не рекомендуется. Лучше воспользоваться прикладным программным интерфейсом API для сохранения глобальных параметров настройки (см. главу 13 первого тома настоящего издания). В данном случае реестр используется лишь для того, чтобы продемонстрировать, каким образом платформенно-ориентированные функции из нетривиального прикладного программного интерфейса API заключаются в оболочку класса Java.

Основным инструментальным средством для работы с реестром является *редактор реестра*. Его применение чревато крупными неприятностями для начинающих пользователей, поэтому для его запуска не предусмотрено никаких визуальных элементов пользовательского интерфейса. Чтобы вызвать редактор реестра из командной строки или в диалоговом окне, которое появляется на экране после выбора пункта меню Пуск⇒Выполнить, следует ввести команду `regedit`. На рис. 12.4 показано, как выглядит окно редактора реестра.

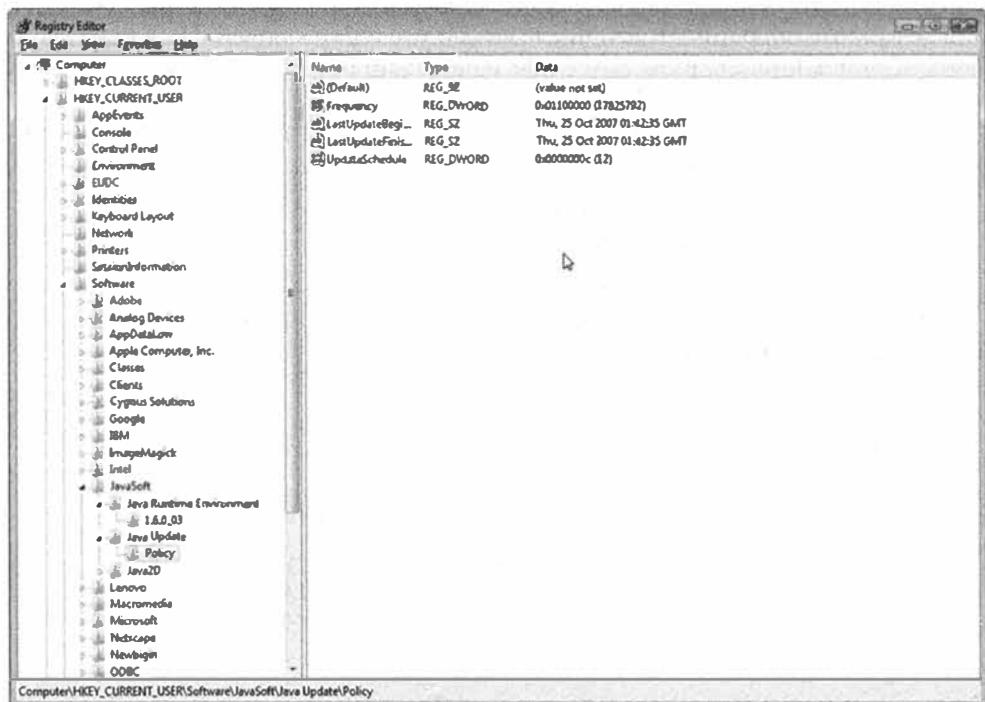


Рис. 12.4. Редактор реестра в Windows

В левой части окна редактора реестра представлены ключи, упорядоченные в виде иерархической древовидной структуры. Каждый ключ находится в поддереве, начинающемся в одном из перечисленных ниже узлов с префиксом **HKEY**.

```
HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
HKEY_LOCAL_MACHINE
```

...

В правой части окна редактора реестра отображаются пары “имя–значение”, связанные с конкретным ключом. Так, если установить версию Java SE 7, то ключ HKEY_LOCAL_MACHINE\Software\JavaSoft\Java Runtime Environment

будет содержать такую пару “имя–значение”:

```
CurrentVersion="1.7.0_10"
```

В данном случае значение представлено в виде символьной строки. Но значения могут быть также представлены в виде целых чисел или массивов байтов.

12.10.2. Интерфейс для доступа к реестру на платформе Java

Рассмотрим создание простого прикладного интерфейса для организации доступа к реестру из прикладной программы на Java, а также его реализацию в платформенно-ориентированном коде. Этот прикладной интерфейс позволяет выполнять лишь несколько операций с реестром. Ради сокращения объема разрабатываемого кода в данном прикладном интерфейсе не поддерживаются многие важные операции, в том числе ввод и удаление ключей. Но для их выполнения в данном прикладном интерфейсе нетрудно реализовать соответствующие функции.

Даже используя ограниченный набор функций, над реестром можно выполнить следующие действия:

- получать все имена, связанные с ключом;
- читать значение, соответствующее имени;
- связывать с именем новое значение.

Для выполнения подобных действий служит следующий класс Java, инкапсулирующий ключ реестра:

```
public class Win32RegKey
{
    public Win32RegKey(int theRoot, String thePath) { . . . }
    public Enumeration names() { . . . }
    public native Object getValue(String name);
    public native void setValue(String name, Object value);

    public static final int HKEY_CLASSES_ROOT = 0x80000000;
    public static final int HKEY_CURRENT_USER = 0x80000001;
    public static final int HKEY_LOCAL_MACHINE = 0x80000002;
    . .
}
```

Метод names() возвращает объект типа Enumeration, который содержит все имена, связанные с данным ключом. Их можно получить с помощью методов hasMoreElements()/nextElement(). Метод getValue() возвращает объект, который может быть символьной строкой, экземпляром класса Integer или байтовым массивом. Параметр value метода setValue() также должен относиться к одному из этих трех типов.

12.10.3. Реализация функций доступа к реестру в виде платформенно-ориентированных методов

В рассматриваемом здесь примере необходимо реализовать следующие действия над реестром Windows:

- получение значения по имени;
- установка значения по имени;
- перечисление имен по ключам.

Правда, ранее уже были рассмотрены все основные инструментальные средства, требующиеся для взаимного преобразования символьных строк и массивов на Java и С. Кроме того, было показано, каким образом генерируются и обрабатываются исключения при возникновении ошибок.

Но по сравнению с примерами из предыдущих разделов применение платформенно-ориентированных методов усложняется двумя обстоятельствами. Во-первых, методы `getValue()` и `setValue()` манипулируют объектами типа `Object`, которые могут относиться к одному из следующих конкретных типов: `String`, `Integer` или `byte[]`. Кроме того, объект перечислимого типа должен сохранять сведения о состоянии между последовательными вызовами методов `hasMoreElements()/nextElement()`. Рассмотрим сначала метод `getValue()`, выполняющий перечисленные ниже действия (его исходный код представлен в листинге 12.22).

1. Открывает ключ в реестре. В прикладном программном интерфейсе API для доступа к реестру требуется, чтобы ключи были открытыми.
2. Запрашивает тип и длину значения, связанного с данным именем.
3. Считывает данные значения в буфер.
4. Вызывает функцию `NewStringUTF()` для создания новой символьной строки с данными значения, если это значение относится к типу `REG_SZ` (т.е. является строкой).
5. Вызывает конструктор класса `Integer`, если это значение относится к типу `REG_DWORD` (т.е. является 32-разрядным целочисленным значением).
6. Вызывает сначала функцию `NewByteArray()` для создания нового байтового массива, а затем функцию `SetByteArrayRegion()` для копирования данных значения в полученный массив, если это значение относится к типу `REG_BINARY` (т.е. является байтовым массивом).
7. Если же это значение не относится ни к одному из перечисленных выше типов или если при вызове функции из прикладного программного интерфейса API возникла какая-нибудь ошибка, генерирует исключение и освобождает все полученные до сих пор ресурсы.
8. Закрывает ключ в реестре и возвращает созданный объект (типа `String`, `Integer` или `byte[]`).

Как видите, рассматриваемый здесь пример наглядно показывает, каким образом формируются объекты разных типов в Java.

В данном платформенно-ориентированном методе совсем не трудно организовать возврат обобщенного значения, поскольку ссылку типа `jstring`, `jint` или `byte[]` можно возвратить просто как ссылку типа `jobject`. Но метод `setValue()` получает ссылку на объект типа `Object` и должен точно определить его конкретный тип, чтобы сохранить его в виде символьной строки, целочисленного значения или байтового массива. Этот тип можно определить, запросив класс объекта `value` и получив ссылки на классы `java.lang.String`, `java.lang.Integer` или `java.lang.byte[]`, а затем сравнив их с помощью функции `IsAssignableFrom()`.

Так, если `class1` и `class2` являются ссылками на два разных класса, то в результате вызова приведенного ниже метода возвратится значение `JNI_TRUE`, при условии, что классы `class1` и `class2` одинаковы или же класс `class1` является подклассом, производным от класса `class2`.

```
(*env)->IsAssignableFrom(env, class1, class2)
```

Но в любом случае ссылки на объекты класса `class1` могут быть приведены к типу `class2`. Так, если приведенный ниже метод возвращает логическое значение `true`, это означает, что объект `value` является байтовым массивом.

```
(*env)->IsAssignableFrom(env, (*env)->GetObjectClass(env, value),  
(*env)->FindClass(env, "[B"))
```

А теперь рассмотрим метод `setValue()`, выполняющий перечисленные ниже действия.

1. Открывает ключ в реестре для записи.
2. Определяет тип записываемого значения.
3. Вызывает функцию `GetStringUTFChars()` для получения указателя на символы, если значение относится к типу `String`.
4. Вызывает метод `intValue()` для извлечения целочисленного значения, хранящегося в объекте-оболочке, если значение относится к типу `Integer`.
5. Вызывает метод `GetByteArrayElements()` для получения указателя на байты, если значение относится к типу `byte[]`.
6. Передает данные и длину значения в реестр.
7. Закрывает ключ в реестре.
8. Освобождает указатель на данные, если речь идет о значении типа `String` или `byte[]`.

И наконец, рассмотрим платформенно-ориентированные методы для перечисления имен по ключам реестра. Как следует из листинга 12.21, эти методы входят в состав класса `Win32RegKeyNameEnumeration`. Чтобы начать процесс перечисления, следует сначала открыть ключ в реестре. На время существования объекта перечислимого типа необходимо сохранить дескриптор ключа. Этот дескриптор относится к типу `DWORD` (32-разрядное целочисленное значение), и поэтому он может быть сохранен в поле `hkey` объекта перечислимого типа. В самом начале процесса перечисления это поле инициализируется с помощью метода `SetIntField()`. А в дальнейшем значение поля `hkey` читается с помощью метода `GetIntField()`.

В объекте перечислимого типа из рассматриваемого здесь примера сохраняются также три других элемента данных. В самом начале процесса перечисления в реестре можно запросить количество пар “имя–значение”, а также длину самого длинного имени. Эти сведения требуются для выделения символьного массива на С и сохраняются в полях count и maxsize объекта перечислимого типа. И наконец, поле index этого объекта инициализируется значением **-1**, указывая на начало процесса перечисления. После инициализации других полей экземпляра в этом поле устанавливается нулевое значение, которое затем инкрементируется на каждом шаге перечисления.

Рассмотрим по очереди действия платформенно-ориентированных методов, поддерживающих перечисление. Начнем с самого простого метода `hasMoreElements()`, который выполняет следующие действия.

1. Извлекает значения полей `index` и `count`.
2. Если в поле `index` оказывается значение **-1**, то вызывается функция `startNameEnumeration()`, которая открывает соответствующий ключ в реестре, запрашивает значение счетчика и максимальную длину, а затем инициализирует поля `hkey`, `count`, `maxsize` и `index`.
3. Если значение в поле `index` меньше значения в поле `count`, возвращает значение `JNI_TRUE`, а иначе – значение `JNI_FALSE`.

Метод `nextElement()` выполняет более сложные, перечисленные ниже действия.

1. Извлекает значения полей `index` и `count`.
2. Если значение в поле `index` равно **-1**, вызывает функцию `startNameEnumeration()`, которая открывает соответствующий ключ в реестре, запрашивает значение счетчика и максимальную длину, а затем инициализирует поля `hkey`, `count`, `maxsize` и `index`.
3. Если значение в поле `index` равно значению в поле `count`, генерирует исключение типа `NoSuchElementException`.
4. Читает следующее имя из реестра.
5. Инкрементирует значение в поле `index`.
6. Если значение в поле `index` снова оказывается равным значению в поле `count`, закрывает ключ в реестре.

Перед компиляцией не забудьте запустить на выполнение утилиту `javah` с обозначениями заголовочными файлами – `Win32RegKey` и `Win32RegKeyNameEnumeration`. Для компилятора от корпорации Microsoft соответствующая команда строка будет выглядеть следующим образом:

```
cl -I jdk\include -I jdk\include\win32 -LD Win32RegKey.c  
advapi32.lib -FeWin32RegKey.dll
```

А в среде `Cygwin` аналогичная команда будет выглядеть так:

```
gcc -mno-cygwin -D __int64="long long" -I jdk\include  
-I jdk\include\win32  
-I c:\cygwin\usr\include\w32api -shared -Wl,--addstdcall-alias  
-o Win32RegKey.dll Win32RegKey.c
```

Но прикладной программный интерфейс API для доступа к реестру рассчитан исключительно на применение в ОС Windows, поэтому в других операционных системах рассматриваемая здесь программа работать не будет.

В листинге 12.23 приведен код программы, позволяющей протестировать новые функции, разработанные для обращения с реестром. Для проведения такого тестирования три пары "имя–значение" (символьная строка, целое число и байтовый массив) сначала добавляются в следующий ключ реестра:

```
HKEY_CURRENT_USER\Software\JavaSoft\Java Runtime Environment
```

Затем перечисляются все имена этого ключа и извлекаются их значения. Ниже приведен результат выполнения данной программы.

```
Default user=Harry Hacker
Lucky number=13
Small primes=2 3 5 7 11 13
```

Несмотря на то что добавление подобных пар имен и значений в данный ключ вряд ли сможет нанести какой-нибудь вред реестру, после выполнения рассматриваемого здесь примера программы их нетрудно удалить, если потребуется, с помощью редактора реестра.

Листинг 12.21. Исходный код из файла win32reg/Win32RegKey.java

```
1 import java.util.*;
2
3 /**
4  * Объект класса Win32RegKey служит для получения и
5  * установки значений в ключах реестра Windows
6  * @version 1.00 1997-07-01
7  * @author Cay Horstmann
8  */
9 public class Win32RegKey
10 {
11     public static final int HKEY_CLASSES_ROOT = 0x80000000;
12     public static final int HKEY_CURRENT_USER = 0x80000001;
13     public static final int HKEY_LOCAL_MACHINE = 0x80000002;
14     public static final int HKEY_USERS = 0x80000003;
15     public static final int HKEY_CURRENT_CONFIG = 0x80000005;
16     public static final int HKEY_DYN_DATA = 0x80000006;
17
18     private int root;
19     private String path;
20
21     /**
22      * Получает значение ключа в реестре
23      * @param name Имя ключа
24      * @return Возвращает значение, связанное с заданным ключом
25      */
26     public native Object getValue(String name);
27
28     /**
29      * Устанавливает значение ключа в реестре
30      * @param name Имя ключа
31      * @param value Новое значение
32      */
33     public native void setValue(String name, Object value);
```

```
34 /**
35  * Создает объект ключа в реестре
36  * @param theRoot Один из следующих ключей:
37  *                 HKEY_CLASSES_ROOT, HKEY_CURRENT_USER,
38  *                 HKEY_LOCAL_MACHINE, HKEY_USERS,
39  *                 HKEY_CURRENT_CONFIG, HKEY_DYN_DATA
40  *
41  * @param thePath Путь к ключу в реестре
42  */
43 public Win32RegKey(int theRoot, String thePath)
44 {
45     root = theRoot;
46     path = thePath;
47 }
48
49 /**
50  * Перечисляет все имена ключей в реестре по пути,
51  * описываемому в данном объекте
52  * @return Возвращает список всех перечисляемых имен
53 */
54 public Enumeration<String> names()
55 {
56     return new Win32RegKeyNameEnumeration(root, path);
57 }
58 static
59 {
60     System.loadLibrary("Win32RegKey");
61 }
62 }
63
64 class Win32RegKeyNameEnumeration implements Enumeration<String>
65 {
66     public native String nextElement();
67     public native boolean hasMoreElements();
68     private int root;
69     private String path;
70     private int index = -1;
71     private int hkey = 0;
72     private int maxsize;
73     private int count;
74
75     Win32RegKeyNameEnumeration(int theRoot, String thePath)
76     {
77         root = theRoot;
78         path = thePath;
79     }
80 }
81
82 class Win32RegKeyException extends RuntimeException
83 {
84     public Win32RegKeyException()
85     {
86     }
87
88     public Win32RegKeyException(String why)
89     {
90         super(why);
91     }
92 }
```

Листинг 12.22. Исходный код из файла win32reg/Win32RegKey.c

```
1  /**
2   * @version 1.00 1997-07-01
3   * @author Cay Horstmann
4   */
5
6  #include "Win32RegKey.h"
7  #include "Win32RegKeyNameEnumeration.h"
8  #include <string.h>
9  #include <stdlib.h>
10 #include <windows.h>
11 JNIEXPORT jobject JNICALL Java_Win32RegKey_getValue(
12     JNIEnv* env, jobject this_obj, jobject name)
13 {
14     const char* cname;
15     jstring path;
16     const char* cpath;
17     HKEY hkey;
18     DWORD type;
19     DWORD size;
20     jclass this_class;
21     jfieldID id_root;
22     jfieldID id_path;
23     HKEY root;
24     jobject ret;
25     char* cret;
26
27     /* получить класс */
28     this_class = (*env)->GetObjectClass(env, this_obj);
29
30     /* получить идентификаторы полей */
31     id_root = (*env)->GetFieldID(env, this_class, "root", "I");
32     id_path = (*env)->GetFieldID(env, this_class, "path",
33                                   "Ljava/lang/String;");
34
35     /* получить поля */
36     root = (HKEY) (*env)->GetIntField(env, this_obj, id_root);
37     path = (jstring) (*env)->GetObjectField(env, this_obj, id_path);
38     cpath = (*env)->GetStringUTFChars(env, path, NULL);
39
40     /* открыть ключ в реестре */
41     if (RegOpenKeyEx(root, cpath, 0, KEY_READ, &hkey)
42         != ERROR_SUCCESS)
43     {
44         (*env)->ThrowNew(env, (*env)->FindClass(env,
45                                         "Win32RegKeyException"),
46                                         "Open key failed");
47         (*env)->ReleaseStringUTFChars(env, path, cpath);
48         return NULL;
49     }
50
51     (*env)->ReleaseStringUTFChars(env, path, cpath);
52     cname = (*env)->GetStringUTFChars(env, name, NULL);
53
54     /* обнаружить тип и длину значения */
55     if (RegQueryValueEx(hkey, cname, NULL, &type, NULL, &size)
```

```
56     != ERROR_SUCCESS)
57 {
58     (*env)->ThrowNew(env, (*env)->FindClass(env,
59                         "Win32RegKeyException"), "Query value key failed");
60     RegCloseKey(hkey);
61     (*env)->ReleaseStringUTFChars(env, name, cname);
62     return NULL;
63 }
64 /* выделить область памяти для хранения конкретного значения */
65 cret = (char*)malloc(size);
66
67 /* прочитать значение */
68 if (RegQueryValueEx(hkey, cname, NULL, &type, cret, &size)
69     != ERROR_SUCCESS)
70 {
71     (*env)->ThrowNew(env, (*env)->FindClass(env,
72                                     "Win32RegKeyException"),
73                                     "Query value key failed");
74     free(cret);
75     RegCloseKey(hkey);
76     (*env)->ReleaseStringUTFChars(env, name, cname);
77     return NULL;
78 }
79
80 /* сохранить значение как символьную строку, целое число или
81 байтовый массив в зависимости от его типа */
82 if (type == REG_SZ)
83 {
84     ret = (*env)->NewStringUTF(env, cret);
85 }
86 else if (type == REG_DWORD)
87 {
88     jclass class_Integer = (*env)->
89         FindClass(env, "java/lang/Integer");
90     /* получить идентификатор метода для конструктора */
91     jmethodID id_Integer = (*env)->GetMethodID(env,
92                                                 class_Integer, "<init>", "(I)V");
93     int value = *(int*) cret;
94     /* вызвать конструктор */
95     ret = (*env)->NewObject(env, class_Integer,
96                             id_Integer, value);
97 }
98 else if (type == REG_BINARY)
99 {
100     ret = (*env)->NewByteArray(env, size);
101     (*env)->SetByteArrayRegion(env, (jarray) ret, 0, size, cret);
102 }
103 else
104 {
105     (*env)->ThrowNew(env, (*env)->FindClass(env,
106                                         "Win32RegKeyException"),
107                                         "Unsupported value type");
108     ret = NULL;
109 }
110
111 free(cret);
112 RegCloseKey(hkey);
```

```
113     (*env)->ReleaseStringUTFChars(env, name, cname);
114
115     return ret;
116 }
117 JNIEXPORT void JNICALL Java_Win32RegKey_setValue(JNIEnv* env,
118                                     jobject this_obj, jstring name, jobject value)
119 {
120     const char* cname;
121     jstring path;
122     const char* cpath;
123     HKEY hkey;
124     DWORD type;
125     DWORD size;
126     jclass this_class;
127     jclass class_value;
128     jclass class_Integer;
129     jfieldID id_root;
130     jfieldID id_path;
131     HKEY root;
132     const char* cvalue;
133     int ivalue;
134
135     /* получить класс */
136     this_class = (*env)->GetObjectClass(env, this_obj);
137
138     /* получить идентификаторы полей */
139     id_root = (*env)->GetFieldID(env, this_class, "root", "I");
140     id_path = (*env)->GetFieldID(env, this_class, "path",
141                                 "Ljava/lang/String;");
142
143     /* получить поля */
144     root = (HKEY)(*env)->GetIntField(env, this_obj, id_root);
145     path = (jstring)(*env)->GetObjectField(env, this_obj, id_path);
146     cpath = (*env)->GetStringUTFChars(env, path, NULL);
147
148     /* открыть ключ в реестре */
149     if (RegOpenKeyEx(root, cpath, 0, KEY_WRITE, &hkey)
150         != ERROR_SUCCESS)
151     {
152         (*env)->ThrowNew(env, (*env)->FindClass(env,
153                                         "Win32RegKeyException"),
154                                         "Open key failed");
155         (*env)->ReleaseStringUTFChars(env, path, cpath);
156         return;
157     }
158
159     (*env)->ReleaseStringUTFChars(env, path, cpath);
160     cname = (*env)->GetStringUTFChars(env, name, NULL);
161
162     class_value = (*env)->GetObjectClass(env, value);
163     class_Integer = (*env)->FindClass(env, "java/lang/Integer");
164     /* определить тип об ъекта, представляющего значение */
165     if ((*env)->IsAssignableFrom(env, class_value, (*env)->
166                                   FindClass(env, "java/lang/String")))
167     {
168         /* это строка, поэтому получить указатель на ее символы */
169         cvalue = (*env)->GetStringUTFChars(env,
```

```
170                                     (jstring) value, NULL);
171     type = REG_SZ;
172     size = (*env)->GetStringLength(env, (jstring) value) + 1;
173 }
174 else if ((*env)->IsAssignableFrom(env,
175                     class_value, class_Integer))
176 {
177     /* это целое значение, поэтому вызвать метод intValue(),
178      чтобы получить значение */
179     jmethodID id_intValue = (*env)->GetMethodID(env,
180                                         class_Integer, "intValue", "()I");
181     ivalue = (*env)->CallIntMethod(env, value, id_intValue);
182     type = REG_DWORD;
183     cvalue = (char*)&ivalue;
184     size = 4;
185 }
186 else if ((*env)->IsAssignableFrom(env, class_value,
187           (*env)->FindClass(env, "[B")))
188 {
189     /* это байтовый массив, поэтому получить указатель на байты */
190     type = REG_BINARY;
191     cvalue = (char*)(*env)->GetByteArrayElements(env,
192                                         (jarray) value, NULL);
193     size = (*env)->GetArrayLength(env, (jarray) value);
194 }
195 else
196 {
197     /* неизвестно, как обрабатывать этот тип данных */
198     (*env)->ThrowNew(env, (*env)->FindClass(env,
199                         "Win32RegKeyException"),
200                         "Unsupported value type");
201     RegCloseKey(hkey);
202     (*env)->ReleaseStringUTFChars(env, name, cname);
203     return;
204 }
205
206 /* установить значение */
207 if (RegSetValueEx(hkey, cname, 0, type, cvalue, size)
208     != ERROR_SUCCESS)
209 {
210     (*env)->ThrowNew(env, (*env)->FindClass(env,
211                         "Win32RegKeyException"),
212                         "Set value failed");
213 }
214
215 RegCloseKey(hkey);
216 (*env)->ReleaseStringUTFChars(env, name, cname);
217
218 /* если значение оказалось символьной строкой или
219  * байтовым массивом, освободить указатель */
220 if (type == REG_SZ)
221 {
222     (*env)->ReleaseStringUTFChars(env, (jstring) value, cvalue);
223 }
224 else if (type == REG_BINARY)
225 {
226     (*env)->ReleaseByteArrayElements(env, (jarray) value,
```

```
227                                         (jbyte*) cvalue, 0);
228 }
229 }
230
231 /* Вспомогательная функция, начинающая процесс
232    перечисления имен по ключам */
233 static int startNameEnumeration(JNIEnv* env,
234                                 jobject this_obj, jclass this_class)
235 {
236     jfieldID id_index;
237     jfieldID id_count;
238     jfieldID id_root;
239     jfieldID id_path;
240     jfieldID id_hkey;
241     jfieldID id_maxsize;
242
243     HKEY root;
244     jstring path;
245     const char* cpath;
246     HKEY hkey;
247     DWORD maxsize = 0;
248     DWORD count = 0;
249
250     /* получить идентификаторы полей */
251     id_root = (*env)->GetFieldID(env, this_class, "root", "I");
252     id_path = (*env)->GetFieldID(env, this_class, "path",
253                                   "Ljava/lang/String;");
254     id_hkey = (*env)->GetFieldID(env, this_class, "hkey", "I");
255     id_maxsize = (*env)->GetFieldID(env, this_class,
256                                      "maxsize", "I");
257     id_index = (*env)->GetFieldID(env, this_class, "index", "I");
258     id_count = (*env)->GetFieldID(env, this_class, "count", "I");
259
260     /* получить значения из полей */
261     root = (HKEY)(*env)->GetIntField(env, this_obj, id_root);
262     path = (jstring)(*env)->GetObjectField(env, this_obj, id_path);
263     cpath = (*env)->GetStringUTFChars(env, path, NULL);
264
265     /* открыть ключ в реестре */
266     if (RegOpenKeyEx(root, cpath, 0, KEY_READ, &hkey)
267         != ERROR_SUCCESS)
268     {
269         (*env)->ThrowNew(env, (*env)->FindClass(env,
270                                               "Win32RegKeyException"),
271                           "Open key failed");
272         (*env)->ReleaseStringUTFChars(env, path, cpath);
273         return -1;
274     }
275     (*env)->ReleaseStringUTFChars(env, path, cpath);
276     /* запросить число и максимальную длину имен */
277     if (RegQueryInfoKey(hkey, NULL, NULL, NULL, NULL, NULL, NULL,
278                         &count, &maxsize, NULL, NULL, NULL) != ERROR_SUCCESS)
279     {
280         (*env)->ThrowNew(env, (*env)->FindClass(env,
281                                               "Win32RegKeyException"),
282                           "Query info key failed");
283     }
     RegCloseKey(hkey);
```

```
284     return -1;
285 }
286
287 /* установить значения в полях x*/
288 (*env)->SetIntField(env, this_obj, id_hkey, (DWORD) hkey);
289 (*env)->SetIntField(env, this_obj, id_maxsize, maxsize + 1);
290 (*env)->SetIntField(env, this_obj, id_index, 0);
291 (*env)->SetIntField(env, this_obj, id_count, count);
292 return count;
293 }
294
295 JNIEXPORT jboolean JNICALL
296         Java_Win32RegKeyNameEnumeration_hasMoreElements(
297                         JNIEnv* env, jobject this_obj)
298 { jclass this_class;
299     jfieldID id_index;
300     jfieldID id_count;
301     int index;
302     int count;
303     /* получить класс */
304     this_class = (*env)->GetObjectClass(env, this_obj);
305
306     /* получить идентификаторы полей */
307     id_index = (*env)->GetFieldID(env, this_class, "index", "I");
308     id_count = (*env)->GetFieldID(env, this_class, "count", "I");
309
310     index = (*env)->GetIntField(env, this_obj, id_index);
311     if (index == -1) /* в первый раз */
312     {
313         count = startNameEnumeration(env, this_obj, this_class);
314         index = 0;
315     }
316     else
317         count = (*env)->GetIntField(env, this_obj, id_count);
318     return index < count;
319 }
320
321 JNIEXPORT jobject JNICALL
322         Java_Win32RegKeyNameEnumeration.nextElement(
323                         JNIEnv* env, jobject this_obj)
324 {
325     jclass this_class;
326     jfieldID id_index;
327     jfieldID id_hkey;
328     jfieldID id_count;
329     jfieldID id_maxsize;
330
331     HKEY hkey;
332     int index;
333     int count;
334     DWORD maxsize;
335
336     char* cret;
337     jstring ret;
338
339     /* получить класс */
340     this_class = (*env)->GetObjectClass(env, this_obj);
```

```
341 /* получить идентификаторы полей */
342 id_index = (*env)->GetFieldID(env, this_class, "index", "I");
343 id_count = (*env)->GetFieldID(env, this_class, "count", "I");
344 id_hkey = (*env)->GetFieldID(env, this_class, "hkey", "I");
345 id_maxsize = (*env)->GetFieldID(env, this_class,
346                               "maxsize", "I");
347
348 index = (*env)->GetIntField(env, this_obj, id_index);
349 if (index == -1) /* first time */
350 {
351     count = startNameEnumeration(env, this_obj, this_class);
352     index = 0;
353 }
354 else
355     count = (*env)->GetIntField(env, this_obj, id_count);
356
357 if (index >= count) /* already at end */
358 {
359     (*env)->ThrowNew(env, (*env)->FindClass(
360                           env, "java/util/NoSuchElementException"),
361                           "past end of enumeration");
362     return NULL;
363 }
364
365 maxsize = (*env)->GetIntField(env, this_obj, id_maxsize);
366 hkey = (HKEY)(*env)->GetIntField(env, this_obj, id_hkey);
367 cret = (char*)malloc(maxsize);
368
369 /* обнаружить следующее имя */
370 if (RegEnumValue(hkey, index, cret, &maxsize, NULL,
371                   NULL, NULL, NULL)
372     != ERROR_SUCCESS)
373 {
374     (*env)->ThrowNew(env, (*env)->FindClass(env,
375                                               "Win32RegKeyException"),
376                                               "Enum value failed");
377     free(cret);
378     RegCloseKey(hkey);
379     (*env)->SetIntField(env, this_obj, id_index, count);
380     return NULL;
381 }
382
383 ret = (*env)->NewStringUTF(env, cret);
384 free(cret);
385
386 /* инкрементировать индекс */
387 index++;
388 (*env)->SetIntField(env, this_obj, id_index, index);
389
390 if (index == count) /* at end */
391 {
392     RegCloseKey(hkey);
393 }
394
395 return ret;
396 }
```

Листинг 12.13. Исходный код из файла win32reg/Win32RegKeyTest.java

```

1 import java.util.*;
2
3 /**
4  * @version 1.02 2007-10-26
5  * @author Cay Horstmann
6  */
7 public class Win32RegKeyTest
8 {
9     public static void main(String[] args)
10    {
11        Win32RegKey key = new Win32RegKey(
12            Win32RegKey.HKEY_CURRENT_USER,
13            "Software\\JavaSoft\\Java Runtime Environment");
14
15        key.setValue("Default user", "Harry Hacker");
16        key.setValue("Lucky number", new Integer(13));
17        key.setValue("Small primes", new byte[] { 2, 3, 5, 7, 11 });
18
19        Enumeration<String> e = key.names();
20
21        while (e.hasMoreElements())
22        {
23            String name = e.nextElement();
24            System.out.print(name + "=");
25
26            Object value = key.getValue(name);
27            if (value instanceof byte[])
28                for (byte b : (byte[]) value)
29                    System.out.print((b & 0xFF) + " ");
30            else
31                System.out.print(value);
32
33            System.out.println();
34        }
35    }
36 }

```

Функции запроса типа данных

- jboolean IsAssignableFrom(JNIEnv* env, jclass cl1, jclass cl2)**

Возвращает значение **JNI_TRUE**, если объекты одного класса (**cl1**) могут быть присвоены объектам другого класса (**cl2**), а иначе — значение **JNI_FALSE**. Проверяет, являются ли классы **cl1** и **cl2** одинаковыми, является ли класс **cl1** производным от класса **cl2**, представляет ли класс **cl2** интерфейс, реализуемый классом **cl1** или же каким-нибудь из его суперклассов.

- jclass GetSuperclass(JNIEnv* env, jclass cl)**

Возвращает суперкласс указанного класса. Если же **cl** представляет класс **Object** или интерфейс, то возвращается значение **NULL**.

Вот и подошел к концу второй том настоящего издания, завершающий долгий экскурс в Java. В этом томе у вас была возможность ознакомиться со многими расширенными средствами программирования на Java и дополнительными прикладными программными интерфейсами API. В начале были рассмотрены такие важные для всякого программирующего на Java темы, как потоки данных, XML, сетевые соединения, базы данных и интернационализация. Затем последовали две длинные главы, посвященные программированию графики и ГПИ. А в завершающих этот том главах обсуждались специальные вопросы безопасности, обработки аннотаций и применения платформенно-ориентированных методов. Надеемся, что этот экскурс в расширенные средства программирования на Java и прикладные программные интерфейсы API оказался для вас занимательным и полезным, и теперь вы сможете применить приобретенные знания и навыки в своих проектах.

Предметный указатель

D

DTD

задаваемые правила, 173
определения
сущностей, 176
типов документов, 171
связывание определений с XML-документами, 172

J

JDBC

активизация трассировки, 292
драйверы
архивные JAR-файлы, 289
типы, 280
место в трехуровневой модели приложений, 282
организация прикладного интерфейса
для доступа к базам данных, 280
основные цели, 281
пул соединений, организация, 343
синтаксис описания источников
данных, 289
спецификация, описание, 281

S

SQL

большие объекты, разновидности, 311
встроенные функции, 287
исключения
анализ, 298
типы, 299
команды
CREATE TABLE, применение, 287
DELETE, применение, 287
INSERT, применение, 287
SELECT, применение, 286
UPDATE, применение, 287
порядок выполнения, 294
метаданные
определение, 327
типы, 327
назначение, 283

переходы

назначение, 313
синтаксис, 313
подготовленные операторы, 305
предупреждения, организация, 299
типы данных
основные, 288
расширенные, 340

X

XML

вывод документов
в виде дерева DOM, 214
средствами StAX, 218
допустимые типы атрибутов, 174
инструкции разметки и обработки, 158
определения DTD
задаваемые правила, 173
способы предоставления, 172
поиск информации средствами
XPath, 195
практическое применение документов,
пример, 182
преобразование документов средствами
XSLT, 225
проверка достоверности документов
на соответствие определениям
DTD, 176
особенности и средства, 171
по схеме типа XML Schema, 181
пространства имен
механизм, 201
 префикс, применение, 202
режим управления, 202
синтаксический анализ
дерева DOM, 161
документов, 159
смешанное содержимое, 157
структуре документов, особенности, 156
сущности, определения и ссылки, 176
схемы документов типа XML Schema
назначение, 179

простые и сложные типы элементов
разметки, 179

сходство и отличие от HTML, 155

формирование документов
без пространств имен, 213
с пространствами имен, 214

чтение документов, 159

элементы разметки и атрибуты,
употребление, 157

XML Schema
назначение и особенности, 171

схемы XML-документов, 179

типы элементов разметки
анонимное определение, 181
простые и сложные, 179

XPath
назначение, 195

операции и выражения, 196

описание узлов дерева DOM, 196

функции, 196

XSLT
назначение, 225

преобразование XML-документов, 225

спецификация, описание, 225

таблицы стилей
создание, 225

шаблоны преобразования, 226

Z

ZIP-архивы
назначение, 91

чтение и запись данных, 91

A

Алгоритмы
безопасного хеширования
SHA, 100
SHA-1, 526

интерполяции, применение, 805

кодирования
Base64, 261
знаком %, 885

компоновки печатаемых страниц, 824

определения краев изображения, 808

регистрации, описание, 512

цифровой подписи, реализация, 525

шифрования
AES, применение, 546
DES, применение, 546

DSA, применение, 529; 530

MD5, применение, 526

RSA, применение, 530; 554
SHA-1, применение, 526

открытым ключом, применение, 553

симметричные, особенности, 553

тайнопись Юлия Цезаря, 482

Аннотации
анализ, средства, 462

в местах употребления типов данных, 454

в объявлениях, 453

для компиляции, назначение, 457

для управления ресурсами,
назначение, 458

инструментальные средства,
назначение, 444

контейнерные, применение, 461

маркерные и однозначные, 452

мета-аннотации, применение, 458

области применения, 443

обозначение в коде, 444

обработка
во время выполнения, 446

на уровне
байт-кода, 446

исходного кода, 446; 461

обработчики событий, 445

объявление, 451

определение, 443

параметры получателей, указание, 456

повторяющиеся, особенности
обработки, 461

процессоры, применение, 461

стандартные, разновидности, 456

Элементы
допустимые типы, 450

объявление, 450

определение, 444

Аплеты
дополнительные полномочия, 525

уровни защиты, 525

Аутентификация
механизмы, 510

модули правил регистрации, 512

пользователей, 510

проблема и разрешение, 534

ролевая, механизм, 516

субъект и принципалы, 512

Б

Базы данных
Apache Derby, 288

- выполнение запросов, 304
 групповые обновления, 338
 заполнение, 301
 запуск сервера, порядок действий, 289
 извлечение автоматически
 генерируемых ключей, 315
 как набор таблиц, 283
 метаданные, применение, 328
 множественные результаты, порядок
 получения, 314
 подготовка запросов, 305
 пул соединений, организация, 343
 разновидности СУБД, 288
 скалярные функции, назначение, 313
 соединение таблиц
 внешнее, 314
 преимущества, 285
 составление запросов
 в текстовом виде на SQL, 285
 по образцу, 285
 схема и каталог, 335
 транзакции для сохранение
 целостности, 337
 хранимые процедуры, 313
 чтение и запись больших объектов, 311
 экспериментальные, создание, 288
- Безопасность**
 верификация байт-кода, 487
 диспетчер защиты
 виды проверок, 491
 назначение, 476
 установка, 498
 источники кода
 кодовая база, 493
 наборы сертификатов, 493
 механизмы обеспечения, 475
 полномочия
 задание, 499
 назначение, 494
 пользовательские, 503
 порядок предположения, 503
 проверка, 495
 правила защиты
 диспетчер, назначение, 496
 назначение, 493
 применение, 492; 497
- Библиотеки**
 ASM, конструирование байт-кодов, 466
 AWT
 всплывающие меню, применение, 890
- расширенные средства, назначение
 и применение, 733
JCE, применение, 552
Swing
 всплывающие меню, 890
 компоненты
 JEditorPane, 682; 686
 JList, 560; 572
 JProgressBar, 688
 JSpinner, 674
 JTable, 576; 607
 JTree, 614; 645
 вспомогательные, применение,
 700; 728
 текстовые, 652
 поддержка перетаскивания, 864
 расширенные средства, 559
- Буфер обмена**
 механизмы
 вырезания и вставки,
 возможности, 846
 передачи данных, возможности, 846
 перетаскивания, поддержка, 861
 передача
 изображений, 853
 объектов, 857
 ссылок на объекты, 860
 текста, 847
 реализация, 846
- Буфера данных**
 определение, 136
 принцип действия, 137
 свойства, 136
- В**
- Ввод-вывод**
 байтов, 64
 двоичных данных, 84
 консольный, особенности, 397
 разновидности, 67
 текста, особенности, 75; 397
- Верификаторы**
 виды проверок, 486
 назначение, 486
- Внутренние фреймы**
 диалоговые окна, создание, 719
 контроль над установкой свойств, 717
 перетаскивание контуров, 719
 расположение
 каскадное, 713
 мозаичное, 715

- создание, 712
состояния, 715
- Время**
местное
обозначение, 349
общие операции, 353
отсчет
по временной шкале в Java, 346
способы и единицы, 346
эпоха, исходная точка отсчета, 346
- по Гринвичу, понятие, 355
- поясное, обозначение, 355
- средства форматирования,
разновидности, 358
- Д**
- Даты**
корректоры
общедоступные, 352
собственные, создание, 353
- местные, обозначение, 349
- средства форматирования,
разновидности, 358
- Двухмерная графика**
внутреннее представление
координат, 737
- возможности рисования в прикладном
интерфейсе Java 2D API, 734
- вывод на печать, порядок действий, 814
- дуги
построение, 739
расчет углов сканирования, 741
- кривые второго и третьего порядка,
построение, 742
- многоугольники, построение, 743
- обводка
по умолчанию, 754
стили линий, 754
- отсечение, назначение, 770
- правила композиции, 773
- преобразования
аффинные, 766
- координат, разновидности, 765
составные, порядок выполнения, 766
- прозрачность, описание в альфа-
канале, 772
- прямоугольники, построение, 738
- указания по воспроизведению
задание, 781
ключи и значения, 781
сглаживание, 783
- участки, построение, операции, 753
фигуры
воспроизведение в конвейере
визуализации, 736
отсечения, 770
- порядок действий при рисовании, 734
- раскраска, разновидности, 762
- Деревья**
воспроизведение узлов, 634
модели
назначение и построение, 615
по умолчанию, 615
специальные, применение, 647
- назначение, 614
- обзначение узлов, 621
- обработка событий, 637
- обход узлов, способы, 632
- основные составляющие и их
обозначение, 614
- поиск узлов по пути к дереву, 624
- пользовательские объекты в узлах, 616
- простые, построение, 615
- редактирование, способы, 624
- редакторы ячеек, реализация, 627
- сворачивание и разворачивание, 620
- Доступ**
к FTP-файлу, защищенному паролем,
особенности, 262
- к базам данных средствами SQL, 283
- к веб-ресурсам, порядок действий, 259
- к веб-странице, защищенной паролем,
порядок действий, 261
- З**
- Загрузчики классов**
в качестве пространств имен,
применение, 480
- иерархия, 477
- инверсия, явление, 479
- контекста, назначение, 479
- разновидности, 477
- специальные, создание, 481
- Запуск настольных приложений,**
поддержка, 884
- И**
- Идентификаторы ресурсов,**
унифицированные
абсолютные и относительные, 258

- иерархическая структура, 258
определение, 257
преобразование адресов, 259
прозрачные и непрозрачные, 258
- Изображения**
выборочные значения цвета пикселя, 798
выбор средств чтения и записи, 788
вывод на печать, 813
манипулирование, 797
размытие, 808
растровые, формирование, 798
фильтрация, 805
цветовые профили ИС, применение, 799
чтение и запись
нескольких изображений, 789
особенности, 787
- Имена ресурсов, унифицированные, 257
- Индикаторы выполнения**
назначение, 688
создание, 689
- Интернационализация**
идентификаторы валют, 377
кодирование символов, 397
комплекты ресурсов, назначение, 400
назначение, 365
окончания строк, интерпретация, 397
прикладной программы, пример, 404
региональные настройки, 366
сортировка, особенности, 385
файлы свойств, применение, 402
форматирование
даты и времени, 378
денежных сумм, 377
чисел, 371
- Интерфейсы**
AnnotatedElement, реализация
и методы, 445
Annotation, расширение и методы, 450
BufferedImageOp, назначение
и реализация, 797; 805
CharSequence, реализация и методы, 52
Comparator, реализация, 386
Compilable, назначение
и реализация, 426
Composite, назначение
и реализация, 774
ContentHandler, методы, 204; 208
DatabaseMetaData
методы, 317; 328
назначение, 328
- DataInput*, реализация и методы, 85
DataOutput, реализация и методы, 84
DataSource, назначение, 342
DiagnosticListener, назначение
и реализация, 432
- Document*, назначение и реализация, 652
Doc, назначение и реализация, 836
Element, реализация и производные, 462
EntityResolver, реализация и метод, 173
EntityResolver, реализация и методы, 178
ErrorHandler, реализация и методы, 177
FileVisitor, назначение, реализация
и методы, 126
- HyperlinkListener*, назначение
и метод, 685
- Invocable*, назначение и реализация, 424
- JavaFileManager*, назначение, 432
- ListCellRenderer*, назначение
и методы, 573
- ListModel*, назначение и методы, 566
- LSSerializer*, применение
и реализация, 215
- Node*
иерархия наследования
и реализация, 160
методы, 203
- Paint*, назначение и реализация, 762
- Path*, назначение и методы, 116
- Printable*, реализация и метод, 814
- PrintRequestAttributeSet*, назначение
и реализация, 814
- Processor*, реализация, 462
- ResultSet*
методы, 295; 317
обработка результирующих
наборов, 295
- ResultSetMetaData*, назначение, 328
- Result*, реализация, 230
- RowSet*
методы, 325
назначение, 323
расширение, 323
- Serializable*, назначение и реализация, 95
- Shape*, назначение и реализация, 737
- Source*, реализация, 229
- Stream*, реализация и методы, 23
- StyledDocument*, назначение
и реализация, 653
- TableCellEditor*, назначение
и методы, 605

TableCellRenderer, реализация и метод, 601
 Tool, назначение и методы, 432
 TreeCellRenderer, реализация и метод, 635
 TreeModelListener, назначение и методы, 645
 TreeSelectionListener, реализация и метод, 637
 XMLReader, реализация, 230
 аннотаций
 назначение, 444
 особенности, 450
 атрибутов печати, назначение, 839
 ввода-вывода
 иерархия, 68
 методы, 69
 реализация, 68
 деревьев, иерархия наследования, 616
 для передачи данных через буфер обмена, назначение, 847
 наборов атрибутов, назначение, 840

K**Классы**

AbstractClassEditor, назначение, 605
 AbstractListModel, назначение и методы, 568
 AbstractSpinnerModel, назначение и методы, 676
 AffineTransform, назначение и методы, 766
 AlphaComposite, назначение и методы, 774
 Area, назначение и методы, 754
 Attributes, методы, 209
 Banner, назначение и методы, 824
 BasicStroke
 конструкторы и методы, 762
 назначение, 754
 Book, назначение и методы, 823
 BufferedImage, назначение и методы, 798
 BufferedReader, назначение и методы, 78
 Buffer, назначение, подклассы и методы, 136
 ByteBuffer, назначение и методы, 130
 ByteLookupTable
 назначение, 807

Charset, назначение и методы, 83
 Cipher, назначение и методы, 546
 ClassLoader, назначение и методы, 481
 Collator, назначение и методы, 386
 Collectors, назначение, методы и коллекторы, 36
 ColorConvertOp, назначение, 807
 ColorModel, назначение и методы, 800
 Color, назначение и методы, 800
 CompilationTask, применение, 433
 ConvolveOp
 конструкторы, 813
 назначение, 807
 Currency, назначение и методы, 377
 DataInputStream
 методы, 72
 назначение, 67
 DataOutputStream, назначение, 67
 DateFormat, назначение и методы, 662
 DateTimeFormatter
 методы, 359
 назначение, 358
 DefaultCellEditor, назначение, 627
 DefaultListModel, назначение и методы, 571
 DefaultMutableTreeNode
 методы, 622; 632
 назначение, 616
 DefaultTreeCellRenderer, назначение и методы, 634
 DefaultTreeModel
 методы, 626
 назначение, 615; 626
 DocumentBuilderFactory, методы, 169; 179; 203
 DocumentBuilder, методы, 169; 177
 DocumentFilter, назначение и методы, 660
 DriverManager
 регистрация драйверов JDBC, 290
 Duration, назначение и методы, 347
 FileChannel, назначение и методы, 138
 FileInputStream, назначение, 71
 FileOutputStream, назначение, 71
 Files
 методы, 117; 123
 назначение, 114
 Graphics
 методы, 736
 назначение, 733

- Graphics2D**
 методы, 734; 736; 754; 766
 назначение, 734
- HashPrintRequestAttributeSet**,
 назначение и методы, 814
- ImageIO**, назначение и методы, 788; 789
- InetAddress**, назначение и методы, 241
- InputStream**
 методы, 64
 назначение, 64
- InputStreamReader**, назначение, 75
- Instant**, назначение и методы, 346
- JavaCompiler**, назначение и методы, 433
- JDesktopPane**, назначение и методы, 713
- JEditorPane**
 методы, 688
 назначение, 684
- JFormattedTextField**
 методы, 661
 назначение, 657
- JInternalFrame**
 методы, 717; 726
 применение, 719
 установка свойств, контроль, 717
- JLabel**, назначение и методы, 574
- JLayer**
 конструкторы и методы, 731
 назначение, 728
- JList**
 методы, 565; 570
 проектный шаблон MVC,
 применение, 566
- JProgressBar**
 методы, 699
 назначение, 688
- JSpinner**
 конструкторы, 680
 методы, 680
- JSplitPane**, конструкторы и методы, 703
- JTabbedPane**, конструкторы и методы,
 704; 709
- JTable**
 методы, 588
 назначение, 577
- JTree**
 конструкторы, 623
 методы, 623; 626; 631; 643
 назначение, 614
- KeyPairGenerator**, применение, 554
- LayerUI**
- конструкторы, 728
методы, 728; 731
- ListResourceBundle**, назначение
и производные, 403
- LocalDateTime**, назначение, 354
- LocalDate**, назначение и методы, 350
- Locale**
 методы, 369
 назначение, 366
- LocalTime**, назначение и методы, 353
- LoginContext**
 методы, 515
 назначение, 511
- LookupOp**, применение, 807
- LookupTable**, назначение, 807
- MaskFormatter**, назначение и методы, 664
- Matcher**, назначение и методы, 148
- MessageDigest**, назначение и методы, 527
- MessageFormat**, назначение
и методы, 392
- NumberFormat**, назначение и методы,
 371; 662
- ObjectOutputStream**, назначение
и методы, 96
- OffsetDateTime**, назначение, 357
- OutputStream**
 методы, 64
 назначение, 64
- OutputStreamWriter**, назначение, 75
- PageFormat**, назначение и методы, 817
- Pattern**, назначение и методы, 23; 150
- PrinterJob**, назначение и методы, 814
- PrintPanel**, назначение и методы, 818
- PrintServiceLookup**, назначение
и методы, 835
- PrintService**, назначение и методы, 835
- PrintWriter**, назначение и методы, 75; 918
- ProgressMonitor**
 методы, 692
 назначение, 692
- ProgressMonitorInputStream**
 конструктор, 695
 назначение, 695
- PushbackInputStream**, назначение, 72
- RandomAccessFile**, назначение
и методы, 87
- Raster**, назначение и методы, 800
- Reader**
 методы, 68
 назначение, 64

- RenderingHints, назначение, 781
RescaleOp
 конструкторы, 812
 назначение, 806
ResourceBundle
 методы, 404
 назначение и производные, 402
RoundRectangle2D, конструкторы, 738
RowFilter, назначение и методы, 592
SAXParserFactory, методы, 207
SAXParser, методы, 208
Scanner, назначение, 77
SecretKeyFactory, применение, 548
SecureRandom, назначение и методы, 548
SecurityManager, назначение
 и методы, 494
ServerSocket, методы, 246
ShortLookupTable
 конструкторы, 813
 назначение, 807
SimpleDoc, назначение и конструктор, 836
Socket
 назначение, 239
SocketChannel
 методы, 257
 назначение, 251
SQLException, назначение и методы, 298
StandardCharsets, назначение
 и константы, 83
Statement, назначение и методы, 294
StreamPrintServiceFactory, назначение, 838
StreamPrintService, назначение, 838
SystemTray, назначение и методы, 889
TableColumn, назначение и методы, 586
TemporalAdjusters, назначение
 и методы, 352
ThreadedEchoHandler, назначение
 и методы, 247
TrayIcon, назначение и методы, 890
TreePath, назначение и методы, 625
URI, назначение и методы, 258
URL
 методы, 257; 265
 назначение, 257
URLConnection
 методы, 260; 265
 назначение, 257
Writer
 методы, 68
 назначение, 64
- XMLInputFactory**, методы, 211
XMLStreamReader, методы, 212
XPathFactory, методы, 200
ZipInputStream, назначение и методы, 91
ZipOutputStream, назначение и методы, 91
ZonedDateTime, назначение и методы, 355
 атрибутов печати, 839
 деревьев, иерархия наследования, 616
 для передачи данных через буфер
 обмена, назначение, 847
 драйверов JDBC, регистрация, 290
 загрузка
 механизм, 476
 разрешение класса, 476
 комплектов ресурсов, определение, 402
 полномочий
 домен защиты, 495
 иерархия, 494
 реализация, 503
 целевые объекты и действия, 499
 потоков
 ввода-вывода, иерархия, 67
 шифрования, применение, 552
 раскраски, назначение, 762
 рисования фигур, иерархия, 737
 сериализация, особенности, 106
 таблиц и столбцов, иерархия, 586
- Кодировки символов**
UTF-8
 преимущество, 82
 применение, 82
UTF-16
 применение, 82
 формы, 82
 в Юникоде
 кодовые единицы, назначение, 68
 кодовые точки, назначение, 81
 назначение, 75
 отметка порядка следования байтов, 398
 частичные, разновидности, 82
- Компилятор Java**
 вызов
 из приложений, потребность, 431
 на компиляцию, простой, 432
 выполнение заданий
 на компиляцию, 432
 динамическое генерирование кода,
 пример, 438
- Комплекты ресурсов**
 иерархии, 401
 назначение, 400

обнаружение, 401
Компоненты Swing
 JEditorPane, назначение и применение, 682
 JList, назначение и применение, 560; 572
 JProgressBar, назначение и применение, 688
 JSpinner, назначение и применение, 674
 JTable, назначение и применение, 576; 607
 JTree, назначение и применение, 614; 645
 вспомогательные, назначение и применение, 700; 728
 источники перетаскивания, реализация, 867
 поддержка перетаскивания, 863
 приемники перетаскивания, реализация, 870
 режимы опускания перетаскиваемых объектов, 871
 текстовые, назначение и применение, 652
Конструирование байт-кодов
 во время загрузки, 472
 порядок действий, 466
Контекст
 воспроизведения шрифтов, применение, 771
 графический, установка, 736
 печатающего устройства, графический, 816

М

Межсетевые адреса и имена хостов, взаимное преобразование, 241 назначение, 277 по протоколу IPv6, 244
Методы сведения, назначение, 29
Многодокументный интерфейс, особенности, 710
Множество Мандельброта, построение, 800
Модели
 баз данных, реляционные, 285
 выбора, назначение и применение, 588 деревьев, назначение и построение, 615 доверительных отношений, 535

печати, назначение, 813
приложений
 клиент-сервер, особенности, 282 трехуровневые, особенности, 282 списков, назначение и реализация, 568 счетчиков, назначение и построение, 675 таблиц, назначение и реализация, 582
Мониторы текущего состояния
 назначение, 692 обработка запросов, 693 потоков ввода, организация, 696 создание, 692

Н

Наборы строк
 кешируемые, 324 применение, 323
Накопление результатов
 в отображениях, 40 в структурах данных, 36
Настольные панели
 назначение, 700 создание, 712
Начальные экраны
 индикация хода загрузки, способы, 879 назначение, 879 отображение, 879
Нисходящие коллекторы
 назначение, 45 разновидности, 45

О

Обработка данных из формы на сервере, порядок действий, 268
Операции
 над моментами и промежутками времени, 347 окончные, выполнение, 29 потоковые, выполнение, 59 сведения, назначение, 49
Отправка электронной почты
 по протоколу SMTP, порядок действий, 276 средствами JavaMail API, 276

П

Пакеты
 com.sun.rowset, назначение, 323
 com.sun.security.auth.module, назначение и состав, 511

- java.awt.datatransfer, назначение, 847
java.net, назначение, 241
java.nio, назначение, 129
java.security, назначение, 525
java.text, назначение, 371
java.time, назначение, 379
java.util.zip, назначение, 131
javax.imageio, назначение, 787
javax.sql.rowset, назначение, 323
javax.sql, назначение, 342
org.w3c.dom, назначение, 159
с интерфейсами аннотаций,
 назначение, 456
- Панели с вкладками**
компоновка, 705
назначение, 704
создание, 704
- Передача данных на веб-сервер
команды и параметры, 268
порядок действий, 269
- Перетаскивание
выполняемые операции,
 разновидности, 861
иницирование операции, 861
источник и приемник, 861
механизм, 861
поддержка в Swing, 864
формы курсора, 862
- Печать
атрибуты
 категории, 841
 наборы, 840
 разновидности, 839
двуухмерной графики, 814
многостраничная, 823
поддерживаемые типы данных, 834
полосовой способ, 816
предварительный просмотр, 825
принцип WYSIWIG, 818
разновидности документов
 определение, 834
 указание, 834
формат бумаги в пунктах, 816
- Платформенно-ориентированный код**
доступ из методов
 к массивам, 926
 к полям экземпляра, 911
 к статическим полям, 914
- инициализация виртуальной машины Java, 933
- методы
альтернативные варианты вызова, 920
кодирование сигнатур, 915
на Java, порядок вызова, 934
объявление, 897
организация вызовов, 897
перегрузка, 898
связывание с программами
 на Java, 902
статические, вызов, 919
строковые параметры, 904
числовые параметры, 902
экземпляра, вызов, 917
- обработка ошибок и исключений, 928
обращение к реестру Windows,
 организация интерфейса, 940
- определение, 895
- порядок обработки, 901
- правила написания, 897
- прекращение работы виртуальной
 машины Java, 934
- соответствие типов
 массивов, 924
- целесообразность применения, 896
- Подписание**
JAR-файлов, процедура, 541
аплетов сертификатами, 541
кода
 на примере аплетов, 542
 особенности, 539
- Полузакрытие**
на стороне клиента, механизм, 250
назначение, 250
- Пользовательский интерфейс на основе**
фреймов, особенности, 710
- Последовательности**
байтов, в потоках ввода-вывода, 64; 81
данных, бесконечные, получение, 23
потоков ввода-вывода, создание, 73
символов, кодировки, 81
- Потоки**
ввода
 определение, 63
 превращение в потоки чтения, 75
 чтение с упреждением, 73
- ввода-вывода
буферизация данных, 73
разделение обязанностей, 71
сериализуемых объектов, 95
сочетание фильтров, 73

типы, 67
 фильтрация, 696
 вывода
 определение, 64
 превращение в потоки записи, 75
 данных
 группирование и разделение
 элементов, 45
 извлечение, 27
 каталогов, применение, 124
 конвейер операций, организация, 21
 назначение, 19
 накопление результатов, 36
 невмешательство, требование, 59
 объектов, 52
 отличия, 21
 параллельные, применение, 57
 преобразование, 26
 примитивных типов, 51
 принцип действия, 20
 соединение, 27
 создание, 22
 сортировка, 28
 упорядочение, 58
 условия нормальной работы, 59
 записи
 автоматическая очистка, 76
 вывод текста, 75
 чтения, ввод текста, 78
 шифрования, применение, 552
Протоколы
HTTP
 запросы и ответы, 262
 формат заголовка, 261
SMTP, назначение, 276
SSL, применение, 557
TCP, особенности, 240
UDP, особенности, 240
Протокольные сообщения, особенности вывода, 398

P

Разделяемые панели
 компоновка, 701
 назначение, 701
 создание, 701
Региональные настройки
 объекты, разновидности, 368
 описание
 дескрипторами, 368
 символьной строкой, 369

определение, 366
 правила составления, 367
Регулярные выражения
 назначение, 141
 применение, 145
 синтаксические конструкции, 141
 флаги сопоставления с шаблоном, 146
Реестр Windows
 назначение, 939
 организация доступа, 940
 редактор, применение, 939
Результирующие наборы
 анализ, 294
 обновляемые, 318
 прокручиваемые, 316

C

Свертка
 операция, назначение, 807
сообщения
 алгоритмы вычисления, 526
 свойства, 526
ядро, 808
Серверы
 обслуживание многих клиентов, 246
 подключение через порт,
 особенности, 237
 типичный цикл работы, 244
Сериализация
объектов
 клонирование, 111
 контроль версий, 109
 механизм, 97
 определение, 95
 формат файлов, 100
 одноэлементных множеств, 108
 переходных полей объектов, 106
Сертификаты
 подписание, 537
 применение, 532
 разработчиков программного
 обеспечения, 540

Синтаксические анализаторы
 XML-документов, разновидности, 159
 древовидные
 DOM-анализатор, применение, 159
 особенности, 159
 определение, 159
 потоковые
 SAX-анализатор, применение, 204
 StAX-анализатор, применение, 209
 особенности, 204

- Синтаксический анализ**
- URI, особенности, 258
 - XML-документов, 159
 - смешанного содержимого, 174
- Системная область**
- вывод уведомлений, 890
 - поддержка, 889
 - назначение, 889
- Системы ZIP-файлов**
- назначение, 127
 - применение, 128
- Слои**
- добавление, 728
 - назначение, 728
 - установка масок событий, 728
- Службы**
- JAAS
 - модули регистрации, назначение, 516
 - назначение, 510
 - применение, 516
 - JNDI, назначение, 342
 - печати
 - обычные, применение, 835
 - потоковые, применение, 838
 - разновидности документов, 834
- Сокеты**
- блокирующие, 252
 - время ожидания, 240
 - закрытие, 244
 - определение, 239
 - открытие, 239
 - прерываемые, 251
- Сортировка**
- ключи, назначение, 387
 - разложение на составляющие, 387
 - режимы, 387
 - уровни избирательности, 386
 - формы нормализации в Юникоде, 387
- Списки**
- ввод и удаление значений, 571
 - воспроизведение ячеек, 572
 - выбор элементов, 562
 - значения прототипных ячеек, 568
 - применение, 568
 - модели, назначение, 568
 - обработка событий, особенности, 562
 - очень длинные, составление, 567
 - простые
 - из символьных строк, 560
 - составление, 560
 - порядок составления, 566
 - расположение элементов, варианты, 561
- Сценарии**
- вызов, 421
 - компиляция, 426
 - механизмы
 - наиболее употребительные, 420
 - получение, 420
 - переадресация ввода-вывода, 423
 - привязки переменных, 422
 - создание, пример, 426
 - функции и методы, вызов, 424
- Счетчики**
- назначение, 674
 - применение, 675
 - создание по модели, 676
- Т**
- Таблицы**
- особенности составления, 577
 - построение по модели, 581
 - простые, составление, 577
 - специальные редакторы ячеек, 581
 - реализация, 605
 - столбцы
 - воспроизведение заголовков, 603
 - выбор, 589
 - доступ, 586
 - изменение размеров, 586
 - порядок воспроизведения, 585
 - сокрытие и показ, 593
 - строки
 - выбор, 588
 - изменение размеров, 588
 - сортировка, 590
 - фильтрация, 591
 - ячейки
 - воспроизведение, 601
 - выбор, 589
 - редактирование, 578; 603
- Текстовые поля**
- ввод целочисленных значений, 658
 - проверка достоверности вводимых данных, 657
 - при потере фокуса ввода, 658
 - средства, применение, 661
 - фильтры документов, применение, 660
 - форматирование
 - специальные средства, применение, 664
 - стандартные средства, применение, 662
- Типы данных**
- Optional

назначение, 31
необязательные значения,
формирование, 33
применение, 31

разновидности
представление, 851
типы MIME, 851
указание, 851

Транзакции
автоматическая фиксация, 337
групповые обновления, 339
точки сохранения, назначение, 338
фиксация и откат, 337

У

Указатели ресурсов, унифицированные, 257

Утилиты

jarsigner, назначение, 533
javah, назначение, 898
javap, применение, 489
keytool, назначение, 530
native2ascii, назначение, 400
policytool, назначение, 502
serialver, назначение, 109
telnet, применение, 236

Ф

Файлы

атрибуты
основные, 121
получение, 121
блокировка
механизм, 138
особенности, 139
разделяемая и исключительная, 139
дополнительные параметры
для выполнения операций, 120
изображений
поддерживаемые форматы, 787
чтение и запись, 787
копирование, перемещение
и удаление, 119
обход по каталогам, 123
отображаемые в памяти
каналы, получение, 129
режимы отображения, 129
правил защиты
места установки, 496
содержимое, 496
создание и распространение, 541

пути
абсолютные и относительные, 114
разрешение, 115
составление, 114

с произвольным доступом
открытие, 87
указатель файла, 87
свойств
назначение, 154
недостатки формата, 154
применение, 402
создание вместе с каталогами, 118
фильтрация по глобальному
шаблону, 124
чтение и запись данных, 117

Форматирование

даты и времени
взаимодействие с унаследованным
кодом, 362
средства, разновидности, 358
стили, 359; 378
элементы шаблонов, 360
текста с учетом региональных настроек,
стили, 379

Форматы

SVG, особенности, 215
XML
назначение, 154
преимущества, 155
выбора, назначение, 395
с обратным порядком следования
байтов, 84
с прямым порядком следования
байтов, 84
файлов для сериализации объектов, 100

Функции JNI

альтернативные варианты вызова
методов, 920
для генерирования исключений, 928
для манипулирования символьными
строками, 904
доступа
к массивам на Java, 926
к полям, 911
особенности вызова, 905

Ц

Цветовые модели

ARGB, назначение, 776
RGB, назначение, 772
SRGB, назначение, 799

- назначение, 799
получение, 800
Центр сертификации
имитация, 537
создание средствами OpenSSL, 538
функции, 537
Цифровые подписи
алгоритмы, 525
верификация, 534
принцип действия, 529
- открытым ключом, процедура, 553
применение, 545
режимы свертывания
и развертывания, 546
схемы заполнения, назначение, 547
хранилища ключей, управление, 531

Ш

- Шифрование**
генерирование ключей, 547
открытый и секретный ключи, 529

Я

- Языки сценариев**
назначение, 420
преимущества и недостатки, 420

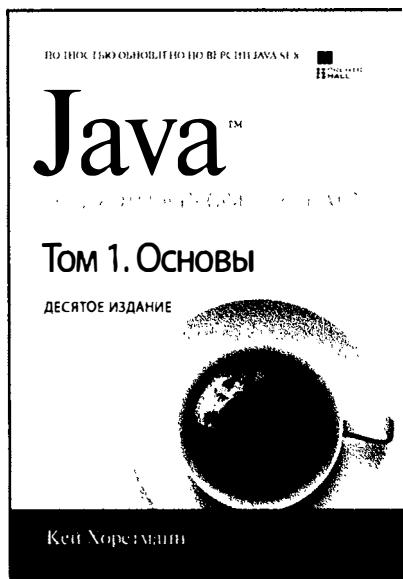
JAVA®

БИБЛИОТЕКА ПРОФЕССИОНАЛА

Том 1. Основы

Десятое издание

Кей Хорстманн



www.williamspublishing.com

Это первый том обновленного, десятого издания исчерпывающего справочного руководства по программированию на Java с учетом всех нововведений в версии Java SE 8. В нем подробно рассматриваются основы программирования на Java, в том числе основные типы и фундаментальные структуры данных, принципы объектно-ориентированного программирования и его реализация в Java, обобщения, коллекции, интерфейсы, лямбда-выражения и функциональное программирование, построение графических пользовательских интерфейсов средствами библиотеки Swing, обработка событий и исключений, развертывание приложений и аплетов, отладка программ, а также параллельное программирование. Излагаемый материал дополняется многочисленными примерами кода, которые не только иллюстрируют основные понятия, но и демонстрируют практические приемы программирования на Java.

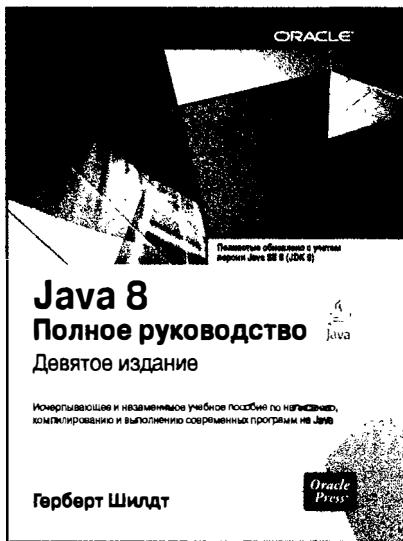
Книга рассчитана на программистов разной квалификации и будет также полезна студентам и преподавателям дисциплин, связанных с программированием на Java.

ISBN 978-5-8459-2084-3 в продаже

JAVA ПОЛНОЕ РУКОВОДСТВО

ДЕВЯТОЕ ИЗДАНИЕ

Герберт Шилдт



www.williamspublishing.com

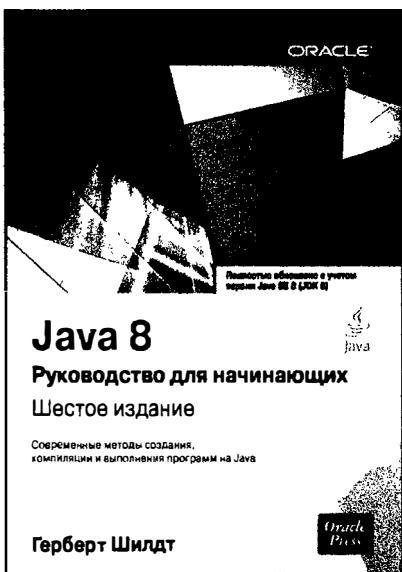
Эта книга является исчерпывающим справочным пособием по языку программирования Java, обновленным с учетом последней версии Java SE 8. В удобной и легко доступной для изучения форме в ней подробно рассматриваются все языковые средства Java, в том числе синтаксис, ключевые слова, операции, управляющие и условные операторы, элементы объектно-ориентированного программирования (классы, объекты, методы, обобщения, интерфейсы, пакеты, коллекции), аплеты и сервлеты, библиотеки классов наряду с такими нововведениями, как стандартные интерфейсные методы, лямбда-выражения, библиотека потоков ввода-вывода, технология JavaFX. Основные принципы и методики программирования на Java представлены в книге на многочисленных и наглядных примерах написания программ. Книга рассчитана на широкий круг читателей, интересующихся программированием на Java.

ISBN 978-5-8459-1918-2

в продаже

Java 8: руководство для начинающих 6-е издание

Герберт Шилдт



www.williamspublishing.com

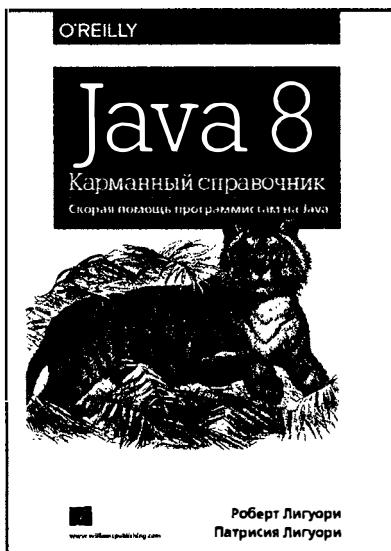
Настоящее, 6-е, издание бестселлера, обновленное с учетом всех новинок последнего выпуска Java Platform, Standard Edition 8 (Java SE 8), позволит новичкам сразу же приступить к программированию на языке Java. Герберт Шилдт, всемирно известный автор множества книг по программированию, уже в начале книги знакомит читателей с тем, как создаются, компилируются и выполняются программы, написанные на языке Java. Далее объясняются ключевые слова, синтаксис и языковые конструкции, образующие ядро Java. Кроме того, в книге рассмотрены темы повышенной сложности: многопоточное программирование, обобщенные типы и средства библиотеки Swing. Не остались без внимания автора и такие новейшие возможности Java SE 8, как лямбда-выражения и методы интерфейсов, используемые по умолчанию. В заключение автор знакомит читателей с JavaFX — новой перспективной технологией создания современных графических интерфейсов пользователя, отличающихся изящным внешним видом и богатым набором элементов управления.

ISBN 978-5-8459-1955-7 в продаже

Java 8

Карманный справочник

*Роберт Лигуори,
Патрисия Лигуори*



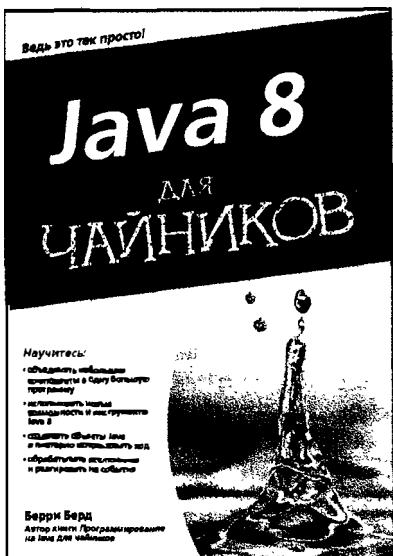
www.williamspublishing.com

Эта небольшая книга, включающая описание новых возможностей Java, до Java SE 8 включительно, будет вашим идеальным спутником, где бы вы ни находились — в офисе, в учебном классе или в пути. Если вам нужно получить оперативные ответы по разработке или отладке программ на Java, то эта книга послужит удобным справочником по стандартным возможностям языка программирования Java и его платформы. Вы найдете здесь полезные примеры программирования, таблицы, рисунки и списки, а также вспомогательную тематическую информацию, в том числе по Java Scripting API, средствам разработки сторонних фирм и основам унифицированного языка моделирования (Unified Modeling Language — UML). Вы узнаете также о новых возможностях Java 8 — лямбда-выражениях и API для работы с датой и временем.

ISBN 978-5-8459-2050-8 в продаже

Java 8 для чайников

Барри Берд



www.dialektika.com

Java — современный объектно-ориентированный язык программирования. Программа, написанная на Java, способна выполняться практически на любом компьютере. Зная Java, можно создавать мощные мультимедийные приложения для любой платформы. Десятки тысяч программистов начинали изучать Java с помощью предыдущих изданий этой книги. Теперь ваша очередь! Независимо от того, на каком языке вы программирували раньше (и даже если вы никогда прежде не программирували), вы быстро научитесь создавать современные кроссплатформенные приложения, используя возможности Java 8.

Основные темы книги:

- ключевые концепции Java;
- грамматика языка;
- повторное использование кода;
- циклы и условные конструкции;
- принципы объектно-ориентированного программирования;
- обработка исключений;
- создание апплетов Java;
- как избежать распространенных ошибок.

ISBN 978-5-8459-1928-1 в продаже

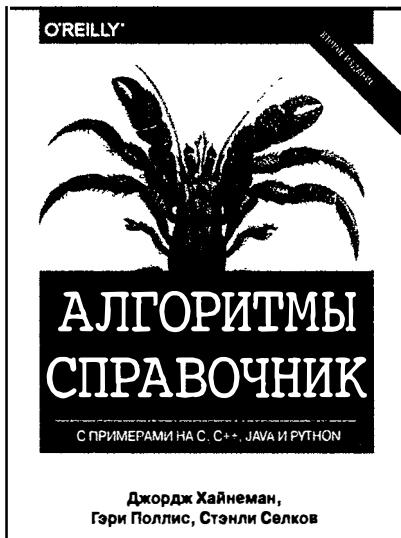
АЛГОРИТМЫ

СПРАВОЧНИК

С ПРИМЕРАМИ НА С, С++, JAVA И PYTHON

ВТОРОЕ ИЗДАНИЕ

Джордж Хайнеман
Гэри Поллис
Стэнли Селков



www.dialektica.com

Если вы считаете, что скорость решения той или иной задачи зависит в первую очередь от мощности компьютера на котором она решается, то эта книга станет для вас откровением с самой первой страницы. Вы узнаете, что наибольший вклад в производительность программы вносят правильно выбранный алгоритм и его реализация в виде компьютерной программы. Выбор подходящего алгоритма среди массы других, способных решить вашу задачу, — дело не самое простое, и этому вы тоже научитесь в данной книге. В новом издании описано множество алгоритмов для решения задач из самых разных областей, и вы сможете выбрать из них и реализовать наиболее подходящий для ваших задач. Здесь даже совершенно незнакомый с математикой читатель найдет все, что нужно для понимания и анализа производительности алгоритма. Написанная профессионалами в своей области, книга достойна занять место на книжной полке любого практикующего программиста.

ISBN 978-5-9908910-7-4 в продаже

АЛГОРИТМЫ НА JAVA

4-Е ИЗДАНИЕ

**Роберт Седжвик,
Кевин Уэйн**



www.williamspublishing.com

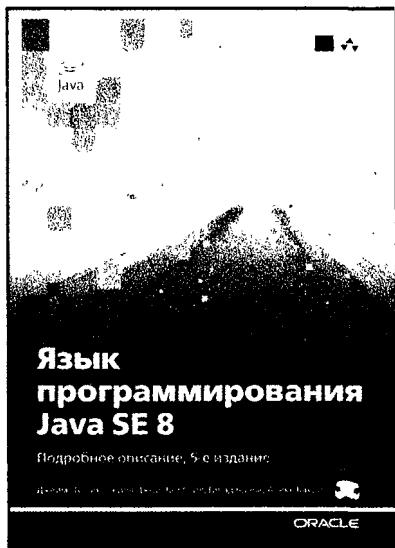
Последнее издание из серии бестселлеров Седжвика, содержащее самый важный объем знаний, наработанных за последние несколько десятилетий. Содержит полное описание структур данных и алгоритмов для сортировки, поиска, обработки графов и строк, включая пятьдесят алгоритмов, которые должен знать каждый программист. В книге представлены новые реализации, написанные на языке Java в доступном стиле модульного программирования — весь код доступен пользователю и готов к применению. Алгоритмы изучаются в контексте важных научных, технических и коммерческих приложений. Клиентские программы и алгоритмы записаны в реальном коде, а не на псевдокоде, как во многих других книгах. Даётся вывод точных оценок производительности на основе соответствующих математических моделей и эмпирических тестов, подтверждающих эти модели.

ISBN 978-5-8459-2049-2 в продаже

ЯЗЫК ПРОГРАММИРОВАНИЯ JAVA SE 8

Подробное описание, 5-е издание

**Джеймс Гослинг,
Билл Джой,
Гай Стил,
Гилад Брача,
Алекс Бакли**



Книга написана разработчиками языка Java и является полным техническим справочником по этому языку программирования. Она обеспечивает полный, точный и подробный охват всех аспектов языка программирования Java. В ней полностью описаны новые возможности, добавленные в Java SE 8, включая лямбда-выражения, ссылки на методы, методы по умолчанию, аннотации типов и повторяющиеся аннотации. В книгу также включено множество поясняющих примечаний. В ней аккуратно обозначены отличия формальных правила языка от практического поведения компиляторов.

www.williamspublishing.com

ISBN 978-5-8459-1875-8 в продаже

Java™

ДЕСЯТОЕ ИЗДАНИЕ

БИБЛИОТЕКА ПРОФЕССИОНАЛА

Том 2. Расширенные средства программирования

В этом надежном, полезном и полностью обновленном по версии Java SE 8 практическом руководстве описаны расширенные языковые средства, библиотеки и прикладные интерфейсы, проиллюстрированные тщательно подобранными и проверенными примерами из практики программирования на Java.

Из второго тома вы узнаете о новых развитых функциональных возможностях, появившихся в версии Java SE 8, в том числе о новых интерфейсах API для потоков данных, даты, времени и календаря, а также о возможностях разработки графических пользовательских интерфейсов, обеспечения безопасности, написания платформенно-ориентированного кода и многое другое. Исходный код всех приведенных примеров обновлен с учетом нововведений в версии профессионала, том 2. *Расширенные средства программирования*, 10-е издание, а их полное описание изящно вплетено в общую канву подробных пояснений расширенных средств программирования на Java. Во втором томе настоящего издания рассматриваются следующие вопросы.

- Применение потоков данных для более эффективной и удобной обработки коллекций
- Эффективный доступ к файлам и каталогам, чтение и запись двоичных и текстовых данных, а также сериализация объектов
- Применение регулярных выражений из пакета, появившегося в версии Java SE 8
- Синтаксический анализ, проверка достоверности данных, формирование XML-документов, применение XPath, XSL и многих других средств обработки данных в формате XML в Java
- Эффективное подключение программ на Java к сетевым службам
- Программирование баз данных средствами JDBC 4.2
- Изящное преодоление трудностей оперирования датами и временем с помощью нового интерфейса API из пакета `java.time`
- Интернационализация прикладных программ с локализованными датами, метками времени, числами, текстом и графическим интерфейсом
- Компиляция и выполнение кода сценариев с помощью специальных интерфейсов API
- Обработка аннотаций
- Повышение безопасности с помощью загрузчиков классов, верификации байт-кода, диспетчеров защиты, установки полномочий и аутентификации пользователей, цифровых подписей, подписания прикладного кода и алгоритмов шифрования
- Овладение расширенными средствами библиотек Swing и AWT для создания списков, таблиц, деревьев, текстовых областей, индикаторов выполнения и прочих элементов графического интерфейса
- Формирование высококачественной графики средствами программного интерфейса Java 2D API
- Эффективное использование кода, написанного на других языках, с помощью платформенно-ориентированных методов из прикладного интерфейса JNI

Подробное рассмотрение основных языковых средств Java, включая объекты, классы, наследование, интерфейсы, события, исключения, графику, основные компоненты библиотек Swing и AWT, обобщения, многопоточную обработку и отладку программ, предлагается в первом томе настоящего издания.

Для тех, у кого имеется опыт программирования на Java и кто собирается перейти к версии Java SE 8, десятое двухтомное издание книги *Java. Библиотека профессионала* послужит надежным, практическим, исчерпывающим руководством для работы, пользующимся больше двадцати лет доверием у разработчиков прикладных программ на платформе Java.

Кей Хорстманн — профессор факультета вычислительной техники в Университете Сан-Хосе, обладатель звания “Чемпион по Java” и частый докладчик на многих отраслевых конференциях. Автор книг *Scala for Impatient* (издательство Addison-Wesley, 2012 г.), *Core Java for the Impatient (Java SE 8. Базовый курс)*, ИД “Вильямс”, 2015 г.), *Java SE 8 for the Really Impatient (Java SE 8. Вводный курс)*, ИД “Вильямс”, 2014 г.), вышедших в издательстве Addison-Wesley. Он написал также более десятка других книг специально для профессиональных программистов и студентов, изучающих дисциплины вычислительной техники.



www.williamspublishing.com



www.informit.com/coreseries

Фото на обложке: Zffoto/Shutterstock.com

ISBN 978-5-9909445-0-3

Категория: программирование
Предмет рассмотрения: язык Java,
версия SE 8
Уровень: промежуточный-
продвинутый



9 785990 944503

ALWAYS LEARNING

PEARSON